

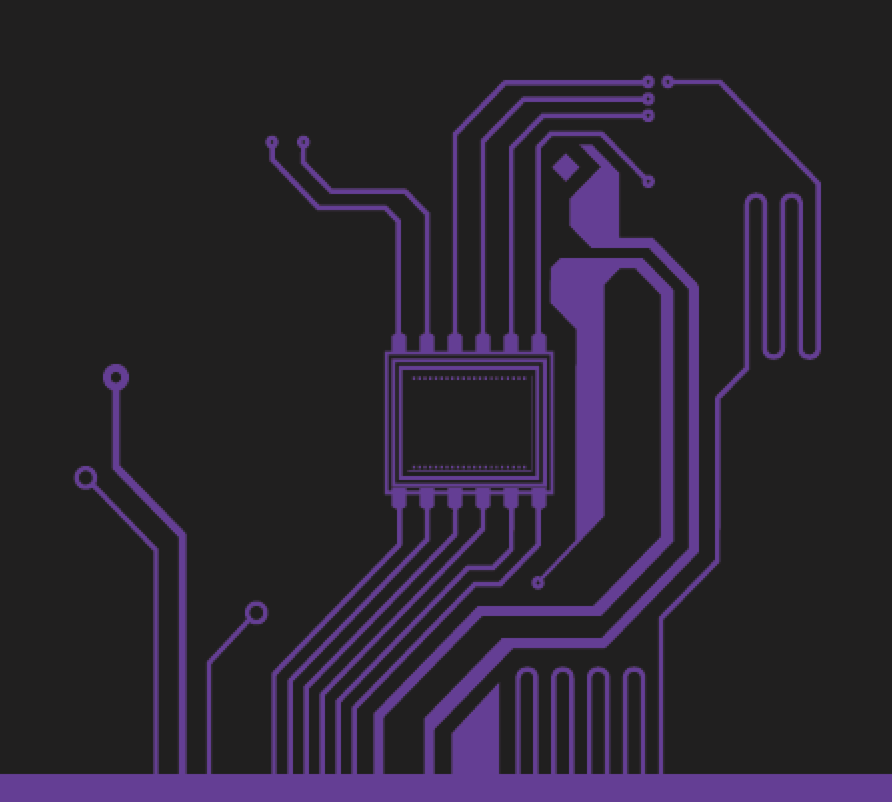

"十二五"普通高等教育
本科国家级规划教材



嵌入式技术基础与实践（第4版）

——ARM Cortex-M0+ KL系列微控制器

◎ 王宜怀 吴瑾 文瑾 著



清华大学出版社

普通高等教育“十二五”国家级规划教材
江苏省高等学校重点教材
电子设计与嵌入式开发实践丛书

嵌入式技术基础与实践(第4版)
——ARM Cortex-M0+KL 系列微控制器

王宜怀　吴　瑾　文　瑾　著

清华大学出版社
北　京

内 容 简 介

本书以恩智浦(NXP)的 ARM Cortex-M0＋内核的 KL 系列微控制器为蓝本阐述嵌入式系统的基本知识要素及软硬件设计方法。全书共 14 章，其中第 1 章为概述，简要阐述嵌入式系统的知识体系、学习误区与学习建议。第 2 章介绍 ARM Cortex-M0＋处理器。第 3 章介绍 KL25/26 存储映像、中断源与硬件最小系统。第 4 章以 GPIO 为例阐述底层驱动概念、设计与应用方法，给出规范的工程组织框架。第 5 章阐述嵌入式硬件构件与底层驱动构件基本规范。第 6 章阐述串行通信接口 UART，并给出第一个带中断的实例。1～6 章囊括学习一个新 MCU 入门环节的完整要素。7～13 章分别介绍了 SysTick、TPM、PIT、LPTMR、RTC、GPIO 的应用实例(键盘、LED 与 LCD)、Flash 在线编程、ADC、DAC、比较器、SPI、I2C、TSI、USB 及其他模块。第 14 章给出了进一步学习指导。

本书提供了网上教学资源，内含所有底层驱动构件源程序、测试实例、文档资料、教学课件及常用软件工具。网上教学资源下载地址：http://sumcu.suda.edu.cn。本书内容还制作了 MOOC，供读者选用。

本书适用于高等学校嵌入式系统的教学或技术培训，也可供 ARM Cortex-M0＋应用工程师作为技术研发参考。

图书在版编目(CIP)数据

嵌入式技术基础与实践：ARM Cortex-M0＋KL 系列微控制器/王宜怀，吴瑾，文瑾著. —4 版. —北京：清华大学出版社，2017
(电子设计与嵌入式开发实践丛书)
ISBN 978-7-302-46757-1

Ⅰ. ①嵌…　Ⅱ. ①王…　②吴…　③文…　Ⅲ. ①微处理器－系统设计　Ⅳ. ①TP332

中国版本图书馆 CIP 数据核字(2017)第 048608 号

责任编辑：魏江江　薛　阳
封面设计：
责任校对：李建庄
责任印制：

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　**邮　　编：**100084
社 总 机：010-62770175　　**邮　　购：**010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
课件下载：http://www.tup.com.cn，010-62795954
印 刷 者：
装 订 者：
经　　销：全国新华书店
开　　本：185mm×260mm　　**印　张：**28.75　　**字　　数：**700 千字
版　　次：2007 年 11 月第 1 版　2017 年 5 月第 4 版　　**印　　次：**2017 年 5 月第 1 次印刷
印　　数：1～　000
定　　价：　.00 元

产品编号：073988-01

前　　言

嵌入式计算机系统简称为嵌入式系统，其概念最初源于传统测控系统对计算机的需求。随着以微处理器(MPU)为内核的微控制器(MCU)制造技术的不断进步，计算机领域在通用计算机系统与嵌入式计算机系统这两大分支分别得以发展。通用计算机已经在科学计算、通信、日常生活等各个领域产生重要影响。在后PC时代，嵌入式系统的广泛应用是计算机发展的重要特征。一般来说，嵌入式系统的应用范围可以粗略地分为两大类：一类是电子系统的智能化(如工业控制、汽车电子、数据采集、测控系统、家用电器、现代农业、传感网应用等)，这类应用也被称为微控制器MCU领域。另一类是计算机应用的延伸(如平板电脑、手机、电子图书等)，这类应用也被称为应用处理器MAP领域。在ARM产品系列中，ARM Cortex-M系列与ARM Cortex-R系列适用于电子系统的智能化类应用，即微控制器领域；ARM Cortex-A系列适用于计算机应用的延伸，即应用处理器领域。不论如何分类，嵌入式系统的技术基础是不变的，即要完成一个嵌入式系统产品的设计，需要有硬件、软件及行业领域相关知识。但是，随着嵌入式系统中软件规模日益增大，对嵌入式底层驱动软件的封装提出了更高的要求，可复用性与可移植性受到特别的关注，嵌入式软硬件构件化开发方法逐步被业界所重视。

2015年12月7日，恩智浦和飞思卡尔完成合并，合并后的公司名称仍为"恩智浦半导体"，成为全球汽车和安全半导体解决方案第一大供应商以及全球第四大非存储类半导体企业。公司持续为互联汽车、物联网设备端到端安全与数据保护等领域提供更为完善的解决方案，旨在帮助人们实现"智慧生活，安全连接"。目前，恩智浦在北京、上海、深圳、苏州等设有办事处或研发中心，在大中华区员工总数超过11 000人。

该公司的微控制器及应用处理器系列，由不同位数、不同封装形式、不同温度范围、所含模块不同等构成了庞大的产品系列，广泛地应用于汽车电子、消费电子、工业控制、网络、无线市场及视频等嵌入式系统各个领域，为嵌入式系统各种应用提供了选择与解决方案，使得用户可以各取所需。不论是电子系统智能化还是计算机应用延伸的嵌入式应用设计，无论需要怎样的系统功能和集成度，总能从这个庞大产品系列中选取一款合适的芯片进行应用开发。这正是嵌入式系统产品设计者所期望的，也节省了嵌入式学习者的时间，可以加快开发进度，提高开发质量。

本书以该公司于2012年开始推出的32位ARM Cortex-M0+内核的KL系列MCU为蓝本阐述嵌入式应用。本书第二版、第三版为普通高等教育"十一五"国家级规划教材，本版为普通高等教育"十二五"国家级规划教材、江苏省高等学校重点教材。本版是在2013年出版的第三版基础上重新撰写。主要变化有：在ARM Cortex-M0+内核不变的前提下，增加了KL26芯片，重新梳理了通用知识要素，优化了底层构件封装；将大部分驱动的使用方法提前阐述，而驱动构件的设计方法后置，目的是先学会使用进行实际编程，后理解构件的设计方法。因构件设计方法部分有一定难度，对于不同要求的教学场景，也可不要求学生理解全部构件的设计方法，讲解一两个即可。

随着作者多年教学与开发的经验积累,本书以嵌入式硬件构件及底层软件构件设计为主线,基于嵌入式软件工程的思想,按照"通用知识—驱动构件使用方法—测试实例—芯片编程结构—构件的设计方法"的路线,逐步阐述电子系统智能化嵌入式应用的软件与硬件设计。

本书具有以下特点。

(1) 把握通用知识与芯片相关知识之间的平衡。书中对于嵌入式"通用知识"的基本原理,以应用为立足点,进行语言简洁、逻辑清晰的阐述,同时注意与芯片相关知识之间的衔接,使读者在更好地理解基本原理的基础上,理解芯片应用的设计,同时反过来,加深对通用知识的理解。

(2) 把握硬件与软件的关系。嵌入式系统是软件与硬件的综合体,嵌入式系统设计是一个软件、硬件协同设计的工程,不能像通用计算机那样,软件、硬件完全分开来看。特别是对电子系统智能化嵌入式应用来说,没有对硬件的理解就不可能写好嵌入式软件,同样没有对软件的理解也不可能设计好嵌入式硬件。因此,本书注重把握硬件知识与软件知识之间的关系。

(3) 对底层驱动进行构件化封装。书中对每个模块均给出根据嵌入式软件工程基本原则并按照构件化封装要求编制底层驱动程序,同时给出详细、规范的注释及对外接口,为实际应用提供底层构件,方便移植与复用,可以为读者进行实际项目开发节省大量时间。

(4) 设计合理的测试用例。书中所有源程序均经测试通过,并保留测试用例在本书的网上光盘中,避免了因例程的书写或固有错误给读者带来烦恼。这些测试用例,也为读者验证与理解带来方便。

(5) 网上教学资源提供了所有模块完整的底层驱动构件化封装程序与测试用例。需要使用PC的程序的测试用例,还提供了PC的C#源程序。网上教学资源中还提供了阅读资料、开发环境的简明使用方法、写入器驱动与使用方法、部分工具软件、有关硬件原理图等。网上教学资源的版本将会适时更新。

(6) 提供硬件核心板、写入调试器,方便读者进行实践与应用。同时提供了核心板与苏州大学恩智浦嵌入式中心设计的扩展板对接,以满足教学实验需要。

本书由王宜怀负责编制提纲和统稿工作,并撰写第1~6章、第14章。吴瑾撰写第7~10章、文瑾撰写第11~13章。研究生王绍丹、徐达、刘锴、陆伟国、司萧俊、白聪、胡唯唯等协助书稿整理及程序调试工作,他们卓有成效的工作,使本书更加实用。恩智浦公司的马莉女士一直关心支持苏州大学恩智浦嵌入式中心的建设,为本书的撰写提供了硬件及软件资料,并提出了许多宝贵建议。恩智浦公司的许多技术人员提供了技术支持。清华大学出版社为本书的出版给予了大力的支持,在此一并表示诚挚的谢意。

鉴于作者水平有限,书中难免存在不足和疏漏之处,恳望读者提出宝贵意见和建议,以便再版时改进。

苏州大学　王宜怀

2017年3月

网上教学资源文件夹结构



该教学资源可在网站 http://sumcu.suda.edu.cn 下载(资源不定期更新)。

- 教学与培训→教学资料
- 嵌入式基础第 4 版→SD－(MO＋)－CD(V2.6)－170401.rar

目　　录

第1章 概　　述

本章导读：作为全书导引，本章阐述了嵌入式系统的基本概念、由来、发展简史、分类及特点；给出嵌入式系统的学习困惑、知识体系及学习建议；介绍了大部分嵌入式系统的核心微控制器——微控制器MCU，以及应用处理器MAP；简要归纳了嵌入式系统的常用术语，以便读者对嵌入式系统基本词汇有初步认识，为后续学习打好基础；简要介绍了嵌入式系统常用的C语言基本语法，以便读者快速了解本书所用C语言知识要素。

1.1 嵌入式系统的定义、发展简史、分类及特点

1.1.1 嵌入式系统的定义

嵌入式系统(Embedded System)是嵌入式计算机系统的简称，有多种多样的定义，但本质是相同的。这里给出美国CMP Books出版的Jack Ganssle和Michael Barr的著作*Embedded System Dictionary*[①]中给出的嵌入式系统定义：**嵌入式系统是一种计算机硬件和软件的组合，也许还有机械装置，用于实现一个特定功能。在某些特定情况下，嵌入式系统是一个大系统或产品的一部分。**世界上第一个嵌入式系统是1971年Busicom公司用Intel单芯片4004微处理器完成的商用计算器系列。该词典还给出了嵌入式系统的一些示例：微波炉、手持电话、计算器、数字手表、录像机、巡航导弹、全球定位系统(Global Positioning System，GPS)接收机、数码相机、传真机、跑步机、遥控器和谷物分析仪等，难以尽数。通过与通用计算机的对比可以更形象地理解嵌入式系统的定义。**该词典给出的通用计算机定义是：计算机硬件和软件的组合，用作通用计算平台。**个人计算机(Personal Computer，PC)是最流行的现代计算机。

再列举其他文献给出的定义，以便了解对嵌入式系统定义的不同表述方式，也可看作从不同角度定义嵌入式系统。

中国《国家标准GB/T 22033—2008 信息技术—嵌入式系统术语》中给出的嵌入式系统定义：**嵌入式系统置入应用对象内部起信息处理和控制作用的专用计算机系统。**它是以应用为中心，以计算技术为基础，软件硬件可剪裁，对功能、可靠性、成本、体积、功耗有严格约束的专用计算机系统，其硬件至少包含一个微控制器或微处理器。

IEEE(国际电机工程师协会)给出的嵌入式系统定义：嵌入式系统是"控制、监视或者辅助装置、机器和设备运行的装置"。

维基百科(英文版)中给出的嵌入式系统定义：嵌入式系统是一种用计算机控制的具有特定功能的较小的机械或电气系统，且经常有实时性的限制，在被嵌入到整个系统中时一般

---

① Jack Ganssle等.英汉双解嵌入式系统词典.马广云等译.北京：北京航空航天大学出版社，2006.

会包含硬件和机械部件。现如今嵌入式系统控制了人们日常生活中的许多设备,98%的微处理器被用在了嵌入式系统中。

国内对嵌入式系统定义曾进行过广泛讨论,有许多不同说法。其中,嵌入式系统定义的涵盖面问题是主要争论焦点之一。例如,有的学者认为不能把手持电话算作嵌入式系统,而只能把其中起控制作用的部分叫嵌入式系统,而手持电话可以称为嵌入式系统的应用产品。其实,这些并不妨碍人们对嵌入式系统的理解,所以不必对定义感到困惑。有些国内学者特别指出,在理解嵌入式系统定义时,不要把嵌入式系统与嵌入式系统产品相混淆。实际上,从口语或书面语言角度,并不区分"嵌入式系统"与"嵌入式系统产品",只要不妨碍对嵌入式系统的理解就没有关系。

总的说来,可以从计算机本身角度概括表述嵌入式系统,那就是:嵌入式系统,即嵌入式计算机系统,它是不以计算机面目出现的"计算机",这个计算机系统隐含在各类具体的产品之中,在这些产品中,计算机程序起到了重要作用。

1.1.2　嵌入式系统的由来及发展简史

1. 嵌入式系统的由来

通俗地说,计算机是因科学家需要一个高速的计算工具而产生的。直到 20 世纪 70 年代,电子计算机在数字计算、逻辑推理及信息处理等方面表现出非凡的能力。在通信、测控与数据传输等领域,人们对计算机技术给予了更大的期待。这些领域的应用与单纯的高速计算要求不同,主要表现在:直接面向控制对象;嵌入到具体的应用体中,而非以计算机的面貌出现;能在现场连续可靠地运行;体积小,应用灵活;突出控制功能,特别是对外部信息的捕捉与丰富的输入输出功能等。由此可以看出,满足这些要求的计算机与满足高速数值计算的计算机是不同的。因此,一种称为微控制器(单片机)①的技术得以产生并发展。为了区分这两种计算机类型,通常把满足海量高速数值计算的计算机称为**通用计算机系统**,而把嵌入到实际应用系统中,实现嵌入式应用的计算机称为**嵌入式计算机系统**,简称嵌入式系统。**可以说,是因为通信、测控与数据传输等领域对计算机技术的需求催生了嵌入式系统的产生。**

2. 嵌入式系统的发展简史

1946 年,诞生了世界上第一台电子数字计算机(The Electronic Numerical Integrator And Calculator,ENIAC),它由美国宾夕法尼亚大学莫尔电工学院制造,重达 30t,总体积约 $90m^3$,占地 $170m^2$,耗电 140kW,运算速度为每秒 5000 次加法,标志着计算机时代的开始。其中,最重要的部件是**中央处理器(Central Processing Unit,CPU),它是一台计算机的运算和控制核心。CPU 的主要功能是解释指令和处理数据,其内部含有运算逻辑部件即算术逻辑运算单元(Arithmetic Logic Unit,ALU)、寄存器部件和控制部件等。**

1971 年,Intel 公司推出了单芯片 4004 微处理器 MPU(Microprocessor Unit),它是世界上第一个商用微处理器,Busicom 公司就是用它制作电子计算器,这就是嵌入式计算机的雏形。1976 年,Intel 公司又推出了 MCS-48 单片机 SCM(Single Chip Microcomputer),这个内部含有 1KB 只读存储器(Read Only Memory,ROM),64B 随机存取存储器(Random

① 微控制器与单片机这两个术语的语义是基本一致的,本书后面除讲述历史之外,一律使用微控制器一词。

Access Memory,RAM)的简单芯片成为世界上第一个单片机,开创了将诸如 ROM、RAM、定时器、并行口、串行口及其他各种功能模块等 CPU 外部资源,与 CPU 一起集成到一个硅片上生产的时代。1980 年,Intel 公司对 MCS-48 单片机进行了完善,推出了 8 位 MCS-51 单片机,并获得巨大成功,开启了嵌入式系统的单片机应用模式。至今,MCS-51 单片机仍有较多应用。这类系统大部分应用于一些简单、专业性强的工业控制系统中,早期主要使用汇编语言编程,后来大部分使用 C 语言编程,一般没有操作系统的支持。

20 世纪 80 年代,逐步出现了 16 位、32 位微控制器(Microcontroller Unit,MCU)。1984 年,Intel 公司推出了 16 位 8096 系列并将其称为嵌入式微控制器,这可能是“嵌入式”一词第一次在微处理机领域出现。这个时期,Motorola、Intel、TI、NXP、Atmel、Microchip、Hitachi、Philips、ST 等公司推出了不少微控制器产品,功能也不断变强,也逐步支持了实时操作系统。

20 世纪 90 年代开始,数字信号处理器(Digital Signal Processing,DSP)、片上系统(System on Chip,SoC)得到了快速发展。嵌入式处理器扩展方式从并行总线型发展出各种串行总线,并被工业界所接受,形成了一些工业标准,如集成电路互连总线(Inter Integrated Circuit,I2C)、串行外设接口(Serial Peripheral Interface,SPI)总线。甚至将网络协议的低两层或低三层都集中到嵌入式处理器上,如某些嵌入式处理器集成了 CAN(Control Area Network)总线接口、以太网接口。随着超大规模集成电路技术的发展,将数字信号处理器 DSP、精简指令集计算机 RISC① 处理器、存储器、I/O、半定制电路集中到单芯片的产品 SoC 中。值得一提的是,ARM 微处理器的出现,较快地促进了嵌入式系统的发展。

21 世纪开始以来,嵌入式系统芯片制造技术快速发展,融合了以太网与无线射频技术,成为物联网(Internet of Things,IoT)关键技术基础。嵌入式系统发展的目标应该是实现信息世界和物理世界的完全融合,构建一个可控、可信、可扩展并且安全高效的信息物理系统(Cyber-Physical Systems,CPS),从根本上改变人类构建工程物理系统的方式。此时的嵌入式设备不但要具备个体智能(Computation,计算)、交流智能(Communication,通信),还要具备在交流中的影响和响应能力(Control,控制与被控),实现“智慧化”。显然,今后嵌入式系统研究要与网络和高性能计算的研究更紧密地合作。

在嵌入式系统的发展历程中,不得不介绍 ARM 公司。由于 ARM 占据了嵌入式市场的最重要份额,本书以 ARM 为蓝本阐述嵌入式应用,下面分单独一小段来简要介绍 ARM。

3. ARM 简介

ARM 即 Advanced RISC Machines 的缩写,既可以认为是一个公司的名称,也可以认为是对一类微处理器的通称,还可以认为是一种技术的名称。

1985 年 4 月 26 日,第一个 ARM 原型在英国剑桥的 Acorn 计算机有限公司诞生,由美

① RISC 是 Reduced Instruction Set Computer(精简指令集计算机)的缩写,其特点是指令数目少、格式一致、执行周期一致、执行时间短、采用流水线技术等。它是 CPU 的一种设计模式,这种设计模式对指令数目和寻址方式都做了精简,使其实现更容易,指令并行执行程度更好,编译器的效率更高。这种设计模式的技术背景是:CPU 实现复杂指令功能的目的是让用户代码更加简洁,但复杂指令通常需要几个指令周期才能实现,且实际使用较少;此外,处理器和主存之间运行速度的差别也变得越来越大。这样,人们发展了一系列新技术,使处理器的指令得以流水执行,同时降低处理器访问内存的次数。RISC 是对比于 CISC(Complex Instruction Set Computer,复杂指令计算机)而言的,可以粗略地认为,RISC 只保留了 CISC 常用的指令,并进行了设计优化,更适合设计嵌入式处理器。

国加州SanJose VLSI技术公司制造。20世纪80年代后期，ARM很快开发成Acorn的台式计算机产品形成了英国的计算机教育基础。1990年成立了Advanced RISC Machines Limited(后来简称为ARM Limited，ARM公司)。20世纪90年代，ARM的32位嵌入式RISC处理器扩展到世界各地，ARM处理器具有耗电少功能强、16位/32位双指令集和众多合作伙伴三大特点。它占据了低功耗、低成本和高性能的嵌入式系统应用领域的重要地位。目前，采用ARM技术知识产权(IP)的微处理器，即通常所说的ARM微处理器，已遍及工业控制、消费类电子产品、通信系统、网络系统、无线系统等各类嵌入式产品市场，基于ARM技术的微处理器的应用，约占据了32位RISC微处理器75%以上的市场份额，ARM技术正在逐步渗入到人们生活的各个方面。但ARM作为设计公司，本身并不生产芯片，而是采用转让许可证制度，由合作伙伴生产芯片。

1993年，ARM公司发布了全新的ARM7处理器核心。其中的代表作为ARM7-TDMI，它搭载了Thumb指令集①，是ARM公司通用32位微处理器家族的成员之一。其代码密度提升了35%，内存占用也与16位处理器相当。

2004年开始，ARM公司在经典处理器ARM11以后不再用数字命名处理器，而统一改用“Cortex”命名，并分为A、M和R三类，旨在为各种不同的市场提供服务。

ARM Cortex-A系列处理器是基于ARMv8A/v7A架构基础的处理器，面向具有高计算要求、运行丰富操作系统以及提供交互媒体和图形体验的应用领域，如智能手机、移动计算平台、超便携的上网本或智能本等。

ARM Cortex-M系列基于ARMv7M/v6M架构基础的处理器，面向对成本和功耗敏感的MCU和终端应用，如智能测量、人机接口设备、汽车和工业控制系统、大型家用电器、消费性产品和医疗器械。

ARM Cortex-R系列基于ARMv7R架构基础的处理器，面向实时系统，为具有严格的实时响应限制的嵌入式系统提供高性能计算解决方案。目标应用包括智能手机、硬盘驱动器、数字电视、医疗行业、工业控制、汽车电子等。Cortex-R处理器是专为高性能、可靠性和容错能力而设计的，其行为具有高确定性，同时保持很高的能效和成本效益。

2009年，推出了体积最小、功耗最低和能效最高的处理器Cortex-M0，这款32位处理器问世后，打破了一系列的授权记录，成了各制造商竞相争夺的香饽饽，仅9个月时间，就有15家厂商与ARM签约。此外，该芯片还将各家厂商拉出了老旧的8位处理器泥潭。2011年，ARM推出了旗下首款64位架构ARMv8。2015年，ARM推出了基于ARMv8架构的一种面向企业级市场的新平台标准。2016年，ARM推出了Cortex-R8实时处理器，可广泛应用于智能手机、平板电脑、物联网领域。

从这里的简单介绍可以看出，不同嵌入式处理器，应用领域有所侧重，开发方法与知识要素也有所不同，基于此，下面介绍嵌入式系统的分类。

1.1.3　嵌入式系统的分类

嵌入式系统的分类标准有很多，有的按照处理器位数来分，有的按照复杂程度来分，还

① Thumb指令集可以看作是ARM指令压缩形式的子集，它是为减小代码量而提出的，具有16位的代码密度。Thumb指令体系并不完整，只支持通用功能，必要时仍需要使用ARM指令，如进入异常时。其指令的格式与使用方式与ARM指令集类似。

有的按其他标准来分，这些分类方法各有特点。从嵌入式系统的学习角度来看，因应用于不同领域的嵌入式系统，其知识要素与学习方法有所不同，所以可以按应用范围简单地把嵌入式系统分为电子系统智能化（微控制器类）和计算机应用延伸（应用处理器）这两大类。一般来说，微控制器与应用处理器的主要区别在于可靠性、数据处理量、工作频率等方面，相对应用处理器来说，微控制器的可靠性要求更高、数据处理量较小、工作频率较低。

1. 电子系统智能化类（微控制器类）

电子系统智能化类的嵌入式系统，主要用于工业控制、现代农业、家用电器、汽车电子、测控系统、数据采集等，这类应用所使用的嵌入式处理器一般被称为**微控制器**（Microcontroller Unit，MCU）。**这类嵌入式系统产品，从形态上看，更类似于早期的电子系统，但内部计算程序起核心控制作用。**这对应于ARM公司的面向各类嵌入式应用的微控制器内核Cortex-M系列及面向实时应用的高性能内核Cortex-R系列。Cortex-R相对于Cortex-M来说，Cortex-R主要针对高实时性应用，如硬盘控制器、网络设备、汽车应用（安全气囊、制动系统、发动机管理）。**从学习与开发角度，电子系统智能化类的嵌入式应用，需要终端产品开发者面向应用对象设计硬件、软件，注重软件、硬件协同开发。因此，开发者必须掌握底层硬件接口、底层驱动及软硬件密切结合的开发调试技能。电子系统智能化类的嵌入式系统，即微控制器，是嵌入式系统的软硬件基础，是学习嵌入式系统的入门环节，且为重要的一环。**从操作系统角度看，电子系统智能化类的嵌入式系统，可以不使用操作系统，也可根据复杂程度及芯片资源的容纳程度，使用操作系统。电子系统智能化类的嵌入式系统使用的操作系统通常是实时操作系统（Real Time Operating System，RTOS），如MQXLite、MQX、FreeRTOS、μCOS-III、μCLinux、VxWorks和eCos等。

2. 计算机应用延伸类（应用处理器类）

计算机应用延伸类的嵌入式系统，主要用于平板电脑、智能手机、电视机顶盒、企业网络设备等，这类应用所使用的嵌入式处理器一般被称为**应用处理器**（Application Processor）。**这类嵌入式系统产品，从形态上看，更接近通用计算机系统。开发方式上，也类似于通用计算机的软件开发方式。从学习与开发角度，计算机应用延伸类的嵌入式应用，终端产品开发者大多购买厂家制作好的硬件实体在嵌入式操作系统下进行软件开发，或者还需要掌握少量的对外接口方式。因此，从知识结构角度，学习这类嵌入式系统，对硬件的要求相对较少。计算机应用延伸类的嵌入式系统，即应用处理器，也是嵌入式系统学习中重要的一环。但是，从学习规律角度看，若是要全面学习掌握嵌入式系统，应该先学习掌握微控制器，然后在此基础上，进一步学习掌握应用处理器编程，而不要倒过来学习。**从操作系统角度看，计算机应用延伸类的嵌入式系统一般使用非实时嵌入式操作系统，通常就称为嵌入式操作系统（Embedded Operation System，EOS），如Android、Linux、iOS、Windows CE等。当然，非实时嵌入式操作系统与实时操作系统也不是明确划分的，只是粗略分类，侧重有所不同而已。现在的RTOS的功能也在不断提升，一般的嵌入式操作系统也在提高实时性。

当然，工业生产车间经常看到利用工业控制计算机、个人计算机（PC）控制机床、生产过程等，这些可以说是嵌入式系统的一种形态，因为它们完成特定的功能，且整个系统不被称为计算机，而是另有名称，如磨具机床、加工平台等。但是，从知识要素角度讲，这类嵌入式系统不具备普适意义，本书不讨论这类嵌入式系统。

1.1.4　嵌入式系统的特点

要谈嵌入式系统的特点,不同学者也许有不同说法。这里从与通用计算机对比的角度来谈嵌入式系统的特点。

与通用计算机系统相比,嵌入式系统的存储资源相对匮乏、速度较低,对实时性、可靠性、知识综合要求较高。嵌入式系统的开发方法、开发难度、开发手段等,均不同于通用计算机程序,也不同于常规的电子产品。嵌入式系统是在通用计算机发展基础上,面向测控系统逐步发展起来的,因此,从与通用计算机对比的角度来认识嵌入式系统的特点,对学习嵌入式系统具有实际意义。

1. 嵌入式系统属于计算机系统,但不单独以通用计算机的面目出现

嵌入式系统的本名叫嵌入式计算机系统(Embedded Computer System),它不仅具有通用计算机的主要特点,又具有自身特点。嵌入式系统也必须要有软件才能运行,但其隐含在种类众多的具体产品中。同时,通用计算机种类屈指可数,而嵌入式系统不仅芯片种类繁多,而且由于应用对象大小各异,嵌入式系统作为控制核心,已经融入到各个行业的产品之中。

2. 嵌入式系统开发需要专用工具和特殊方法

嵌入式系统不像通用计算机那样有了计算机系统就可以进行应用软件的开发。一般情况下,微控制器或应用处理器芯片本身不具备开发功能,必须要有一套与相应芯片配套的开发工具和开发环境。这些工具和环境一般基于通用计算机上的软硬件设备以及逻辑分析仪、示波器等。开发过程中往往有工具机(一般为PC或笔记本)和目标机(实际产品所使用的芯片)之分,工具机用于程序的开发,目标机作为程序的执行机,开发时需要交替结合进行。编辑、编译、链接生成机器码在工具机完成,通过写入调试器将机器码下载到目标机中,进行运行与调试。

3. 使用MCU设计嵌入式系统,数据与程序空间采用不同存储介质

在通用计算机系统中,程序存储在硬盘上。实际运行时,通过操作系统将要运行的程序从硬盘调入内存(RAM),运行中的程序、常数、变量均在RAM中。而在以MCU为核心的嵌入式系统中,一般情况下,其程序被固化到非易失性存储器中①。变量及堆栈使用RAM存储器。

4. 开发嵌入式系统涉及软件、硬件及应用领域的知识

嵌入式系统与硬件紧密相关,嵌入式系统的开发需要硬件、软件协同设计、协同测试。同时,由于嵌入式系统专用性很强,通常是用在特定应用领域,如嵌入在手机、冰箱、空调、各种机械设备、智能仪器仪表中起核心控制作用,功能专用,因此,进行嵌入式系统的开发,还需要对领域知识有一定的理解。当然,一个团队协作开发一个嵌入式产品,其中各个成员可以扮演不同角色,但对系统的整体理解与把握并相互协作,有助于一个稳定可靠的嵌入式产品的诞生。

---

①　目前,非易失性存储器通常为Flash存储器,特点见第9章。

1.2 嵌入式系统的学习困惑、知识体系及学习建议

1.2.1 嵌入式系统的学习困惑

关于嵌入式系统的学习方法，因学习经历、学习环境、学习目的、已有的知识基础等不同，可能在学习顺序、内容选择、实践方式等方面有所不同。但是，应该明确哪些是必备的基础知识，哪些应该先学，哪些应该后学；哪些必须通过实践才能获得；哪些是与具体芯片无关的通用知识，哪些是与具体芯片或开发环境相关的知识。

嵌入式系统初学者应该通过选择一个具体MCU作为蓝本，期望通过学习实践，获得嵌入式系统知识体系的通用知识，**其基本原则是：入门时间较快、硬件成本较少，软硬件资料规范、知识要素较多，学习难度较低。**

由于微处理器与微控制器种类繁多，也可能由于不同公司、不同机构出于自身的利益，给出一些误导性宣传，特别是我国芯片制造技术的落后及其他相关情况，使得人们对微控制器及应用处理器的发展，在认识与理解上存在差异，使得一些初学者有些困惑。下面简要分析初学者可能存在的几个困惑。

(1) 嵌入式系统学习困惑1——选择入门芯片：是微控制器还是应用处理器？

在了解嵌入式系统分为微控制器与应用处理器两大类之后，入门芯片选择的困惑表述为：**选微控制器，还是应用处理器作为入门芯片？**从性能角度，与应用处理器相比，微控制器工作频率低、计算性能弱、稳定性高、可靠性强。从使用操作系统角度，与应用处理器相比，开发微控制器程序一般使用RTOS，也可以不使用操作系统，而开发应用处理器程序，一般使用非实时操作系统。从知识要素角度，与应用处理器相比，开发微控制器程序一般更需要了解底层硬件，而开发应用处理器终端程序，一般是在厂家提供的驱动基础上基于操作系统开发，更像开发一般PC软件方式。从这段分析可以看出，**要想成为一名知识结构合理且比较全面的嵌入式系统工程师，应该选择一个较典型的微控制器作为入门芯片，且从不带操作系统(No Operating System，NOS)学起，由浅入深，逐步推进。**

关于学习芯片的选择还有一个困惑，就是系统的工作频率。误认为选择工作频率高的芯片进行入门学习，表示更先进。实际上，工作频率高可能给初学者带来学习过程中的不少困难。

实际嵌入式系统设计不是追求芯片计算速度、工作频率、操作系统等因素，而是追求稳定可靠、维护、升级、功耗、价格等指标。

(2) 嵌入式系统学习困惑2——选择操作系统：NOS、RTOS或EOS。

操作系统选择的困惑表述为：**开始学习时，是无操作系统(NOS)、实时操作系统(RTOS)，还是一般嵌入式操作系统(EOS)？**

学习嵌入式系统的目的是为了开发嵌入式应用产品，许多人想学习嵌入式系统，不知道该从何学起，具体目标也不明确。于是，看了一些培训广告，看了书店中书架上种类繁多的嵌入式系统的书籍，或上网以“嵌入式系统”为关键词进行查询，然后参加培训或看书，开始“学习起来”。一些初学者，往往选择一个嵌入式操作系统就开始学习了。不十分恰当的比

喻，有点儿像“瞎子摸大象”，只了解其一个侧面。这样难以对嵌入式产品的开发过程有个全面了解。**针对许多初学者选择“×××嵌入式操作系统+×××处理器”的嵌入式系统入门学习模式，本书认为是不合适的。本书的建议是：首先把嵌入式系统软件与硬件基础打好了，再根据实际应用需要，选择一种实时操作系统(RTOS)进行实践。**我们必须明确认识到，RTOS是开发某些嵌入式产品的辅助工具，是手段，不是目的。况且一些小型微型嵌入式产品并不需要RTOS。所以，一开始就学习RTOS，并不符合“由浅入深、循序渐进”的学习规律。

另外一个问题是选RTOS，还是EOS？面向测控领域的一般选择RTOS，例如MQX、MQXLite、FreeRTOS、μCOS-III、μCLinux、VxWorks和eCos等。本书建议是MQX及MQXLite[①]。实际上RTOS种类繁多，实际使用何种RTOS，一般需要工作单位确定。基础阶段主要学习RTOS的基本原理，并学习在RTOS之上的软件开发方法，而不是学习如何设计RTOS。面向平板电脑、智能手机、电视机顶盒、企业网络设备编程，一般选择EOS，如Android、Linux、Windows CE等，可根据实际需要进行学习。

对于嵌入式操作系统，一定不要一开始就学，这样会走很多弯路，也会使自己对嵌入式系统感到畏惧。等软件硬件基础打好了，再学习就感到容易理解。实际上，众多MCU嵌入式应用，并不一定需要操作系统或只需一个小型RTOS。也可以根据实际项目需要再学习特定的RTOS。一定不要被一些嵌入式实时操作系统培训班宣传所误导，而忽视实际嵌入式系统软件硬件基础知识的学习。不论如何，以开发实际嵌入式产品为目标的学习者，不要把过多的精力花在设计或移植RTOS、EOS上面。正如很多人使用Windows操作系统，而设计Windows操作系统只有Microsoft。许多人“研究”Linux，但从来没有使用它开发过真正的嵌入式产品，浪费了时间，人的精力是有限的，学习必须有所选择。有的学习者，学了很长时间的嵌入式操作系统移植，而不进行实际嵌入式系统产品的开发，到了最后，做不好一个稳定的嵌入式系统小产品，偏离了学习目标，甚至放弃了嵌入式系统领域。

(3) 嵌入式系统学习困惑3——硬件与软件：如何平衡？

以MCU为核心的嵌入式技术的知识体系必须通过具体的MCU来体现、实践与训练。但是，选择任何型号的MCU，其芯片相关的知识只占知识体系的20%左右，80%左右是通用知识。但是这80%的通用知识，必须通过具体实践才能进行，所以学习嵌入式技术要选择一个系列的MCU。但不论如何，嵌入式系统均含有硬件与软件两大部分，它们之间的关系如何呢？

有些学者，仅从电子角度认识嵌入式系统，认为“嵌入式系统=MCU硬件系统+小程序”。这些学者，大多具有良好的电子技术基础知识。实际情况是，早期MCU内部RAM小、程序存储器外接，需要外扩各种I/O，没有像现在这样USB、嵌入式以太网等较复杂的接口，因此，程序占总设计量小于50%，使人们认为嵌入式系统(MCU)是“电子系统”，以硬件为主、程序为辅。但是，随着MCU制造技术的发展，不仅MCU内部RAM越来越大，Flash进入MCU内部改变了传统的嵌入式系统开发与调试方式，固件程序可以被更方便地调试与在线升级，许多情况与开发PC程序的难易程度相差无几，只不过开发环境与运行环境不

---

① 王宜怀等.嵌入式实时操作系统MQX应用开发技术——ARM Cortex-M微处理器.北京：电子工业出版社，2014.

是同一载体而已。这些情况使得嵌入式系统的软硬件设计方法发生了根本变化。特别是因软件危机而发展起来的软件工程学科对嵌入式系统软件的发展也产生重要影响,产生了嵌入式系统软件工程。

有些学者,仅从软件开发角度认识嵌入式系统,甚至有的仅从嵌入式操作系统角度认识嵌入式系统。这些学者,大多具有良好的计算机软件开发基础知识,认为硬件是生产厂商的事,没有认识到,嵌入式系统产品的软件与硬件均是需要开发者设计的。本书作者常常接到一些关于嵌入式产品稳定性的咨询电话,发现大多数是由于软件开发者对底层硬件的基本原理不理解造成的。特别是,有些功能软件开发者,过分依赖于底层硬件的驱动软件设计完美,自己对底层驱动原理知之甚少。实际上,一些功能软件开发者,名义上是在做嵌入式软件,但仅仅是使用嵌入式编辑、编译环境与下载工具而已,本质与开发通用PC软件没有两样。而底层硬件驱动软件的开发,若不全面考虑高层功能软件对底层硬件的可能调用,也会使得封装或参数设计得不合理或不完备,导致高层功能软件的调用困难。从这段描述可以看出,若把一个嵌入式系统的开发孤立地分为硬件设计、底层硬件驱动软件设计、高层功能软件设计,一旦出现了问题,就可能难以定位。**实际上,嵌入式系统设计是一个软件、硬件协同设计工程,不能像通用计算机那样,软件、硬件完全分开来看,要在一个大的框架内协调工作**。在一些小型公司,需求分析、硬件设计、底层驱动、软件设计、产品测试等过程可能是由同一个团组完成,这就需要团队成员,对软件、硬件及产品需求有充分认识,才能协作开发好。甚至许多实际情况是在一些小公司这个"团队"可能就是一个人。

面对学习嵌入式系统以软件为主还是以硬件为主,或是如何选择切入点,如何在软件与硬件之间取得一些平衡,对于这个困惑的建议是:**要想成为一名真正的嵌入式系统设计师,在初学阶段,必须重视打好嵌入式系统的硬件与软件基础**。以下是从事嵌入式系统设计二十多年的一个美国学者 John Catsoulis 在 *Designing Embedded Hardware* 一书中关于这个问题的总结:**嵌入式系统与硬件紧密相关,是软件与硬件的综合体,没有对硬件的理解就不可能写好嵌入式软件,同样没有对软件的理解也不可能设计好嵌入式硬件**。

充分理解嵌入式系统软件与硬件的相互依存关系,对嵌入式系统的学习有良好的促进作用。一方面,既不能只重视硬件,而忽视编程结构、编程规范、软件工程的要求、操作系统等知识的积累;另一方面,也不能仅从计算机软件角度,把通用计算机学习过程中的概念与方法生搬硬套到嵌入式系统的学习实践中,忽视嵌入式系统与通用计算机的差异。在嵌入式系统学习与实践的初始阶段,应该充分了解嵌入式系统的特点,根据自身已有的知识结构,制定适合自身情况的学习计划。**目标应该是打好嵌入式系统的硬件与软件基础,通过实践,为成为良好的嵌入式系统设计师建立起基本知识结构**。学习过程,可以通过具体应用系统为实践载体,但不能拘泥于具体系统,应该有一定的抽象与归纳。例如,有的初学者开发一个实际控制系统,没有使用实时操作系统,但不要认为实时操作系统不需要学习,要注意知识学习的先后顺序与时间点的把握。又例如,有的初学者以一个带有实时操作系统的样例为蓝本进行学习,但不要认为,任何嵌入式系统都需要使用实时操作系统,甚至把一个十分简明的实际系统加上一个不必要的实时操作系统。因此,**片面认识嵌入式系统,可能导致学习困惑**。应该根据实际项目需要,锻炼自己分析实际问题、解决问题的能力。这是一个较长期的需要静下心来的学习与实践过程,不能期望通过短期培训完成整体知识体系的建立,应该重视自身实践,全面地理解与掌握嵌入式系统的知识体系。

1.2.2　嵌入式系统的知识体系

从由浅入深、由简到繁的学习规律来说，嵌入式学习的入门应该选择微控制器，而不是应用处理器，应通过对微控制器基本原理与应用的学习，逐步掌握嵌入式系统的软件与硬件基础，然后在此基础上进行嵌入式系统其他方面知识的学习。

本书主要阐述以MCU为核心的嵌入式技术基础与实践。**要完成一个以MCU为核心的嵌入式系统应用产品设计，需要有硬件、软件及行业领域相关知识。硬件主要有MCU的硬件最小系统、输入/输出外围电路、人机接口设计。软件设计有固化软件的设计，也可能含PC软件的设计。行业知识需要通过协作、交流与总结获得。**

概括地说，学习以MCU为核心的嵌入式系统，需要以下软件硬件基础知识与实践训练，即以MCU为核心嵌入式系统的基本知识体系如下[①]。

(1) **掌握硬件最小系统与软件最小系统框架。** 硬件最小系统是包括电源、晶振、复位、写入调试器接口等可使内部程序得以运行的、规范的、可复用的核心构件系统[②]。软件最小系统框架是一个能够点亮一个发光二极管的，甚至带有串口调试构件的，包含工程规范完整要素的可移植与可复用的工程模板[③]。

(2) **掌握常用基本输出的概念、知识要素、构件使用方法及构件设计方法。** 如通用I/O(GPIO)、模数转换AD、数模转换DA、定时器模块等。

(3) **掌握若干嵌入式通信的概念、知识要素、构件使用方法及构件设计方法。** 如串行通信接口UART、串行外设接口SPI、集成电路互连总线I2C、CAN、USB、嵌入式以太网、无线射频通信等。

(4) **掌握常用应用模块的构件设计方法及使用方法及数据处理方法。** 如显示模块(LED、LCD、触摸屏等)、控制模块(控制各种设备，包括PWM等控制技术)等。数据处理如图形、图像、语音、视频等处理或识别等。

(5) **掌握一门实时操作系统RTOS的基本用法与基本原理。** 作为软件辅助开发工具的实时操作系统RTOS，也可以作为一个知识要素。可以选择一种实时操作系统RTOS(如MQXLite、MQX、VxWorks、μC/OS、μCLinux、QNX、eCOS)进行学习实践，没有必要在没有明确目的的情况下，选择几种同时学习。学好一种，在确有必要使用另一种RTOS时，再学习，也可触类旁通。

(6) **掌握嵌入式软硬件的基本调试方法。** 如断点调试、打桩调试、printf调试方法等。在嵌入式调试过程中，特别要注意确保在正确硬件环境下调试未知软件，在正确软件环境下调试未知硬件。

这里给出的是基础知识要素，关键还是看如何学习，是他人做好了驱动程序你使用，还是你自己完全掌握知识要素，从底层开始设计驱动程序，同时熟练掌握驱动程序的使用。这体现在不同层面的人才培养中。而应用中的硬件设计、软件设计、测试等都必须遵循嵌入式软件工程的方法、原理与基本原则。所以，嵌入式软件工程也是嵌入式系统知识体系的有机

---

① 有关名词解释见1.4节，本书将逐步学习这些内容。

② 将在第3章阐述。

③ 将在第4章、第6章阐述。

组成部分，只不过，它融于具体项目的开发过程之中。

若是主要学习应用处理器类嵌入式应用，也应该在了解 MCU 知识体系基础上，选择一种嵌入式操作系统（如 Android、Linux、Windows CE 等）进行学习实践。目前 APP 开发也是嵌入式应用的一个重要组成部分，可选择一种 APP 开发进行实践（如 Android APP、iOS APP、Windows Phone APP 等）。

与此同时，在 PC 上，利用面向对象编程语言进行测试程序、网络侦听程序、Web 应用程序的开发及对数据库的基本了解与应用，也应逐步纳入嵌入式应用的知识体系中。此外，理工科的公共基础，本身就是学习嵌入式系统的基础。

1.2.3 基础阶段的学习建议

十多年来，我们逐步探索与应用构件封装原则，把硬件相关的部分封装成底层构件，统一接口，努力使高层程序与芯片无关，可以在各种芯片应用系统移植与复用，试图降低学习难度。学习的关键就变成基本了解底层构件设计方法，掌握底层构件的使用方式，在此基础上，进行嵌入式系统设计与应用开发。当然，掌握底层构件的设计方法，学会实际设计一个芯片的某一模块的底层构件，也是本科学生应该掌握的基本知识。对于专科类学生，可以直接使用底层构件进行应用编程，但也需了解知识要素的抽取方法与底层构件基本设计过程。对于看似庞大的嵌入式系统知识体系，可以使用“电子札记”的方式进行知识积累与补缺补漏，任何具有一定理工科基础的学生，通过一段稍长时间的静心学习与实践，都能学好嵌入式系统。

下面针对嵌入式系统的学习困惑，从嵌入式系统的知识体系角度，对广大渴望学习嵌入式系统的学子提出 5 点基础阶段的学习建议。

（1）遵循“先易后难，由浅入深”的原则，打好软硬件基础。跟随本书，充分利用本书提供的软硬件资源及辅助视频材料，逐步实验与实践[①]；充分理解硬件基本原理、掌握功能模块的知识要素、掌握底层驱动构件的使用方法、掌握一两个底层驱动构件的设计过程与方法；熟练掌握在底层驱动构件基础上，利用 C 语言编程实践。理解学习嵌入式系统，必须勤于实践。关于汇编语言问题，随着 MCU 对 C 编译的优化支持，可以只了解几个必需的汇编语句，但必须通过第一个程序理解芯片初始化过程、中断机制、程序存储情况等区别于 PC 程序的内容；最好认真理解一个真正的汇编实例。另外，为了测试的需要，最好掌握一门 PC 上面向对象编程高级语言（如 C#），本书网上资源给出了 C# 快速入门方法与实例。

（2）充分理解知识要素、掌握底层驱动构件的使用方法。本书对诸如 GPIO、UART、定时器、PWM、AD、DA、Flash 在线编程、USB 等模块，首先阐述其通用知识要素，随后给出其底层驱动构件的基本内容。期望读者在充分理解通用知识要素基础上，学会底层驱动构件使用方法。即使这一点，也要下一番功夫。俗话说，书读百遍，其义自见。有关知识要素涉及硬件基本原理，以及对底层驱动接口函数功能及参数的理解，需反复阅读、反复实践，查找资料，分析、概括及积累。对于硬件，只要在深入理解 MCU 的硬件最小系统基础上，对上述

---

① 这里说的实验主要指通过重复或验证他人的工作，其目的是学习基础知识，这个过程一定要经历。实践是自己设计，有具体的“产品”目标。如果能花 500 元左右自己做一个具有一定功能的小产品，且能稳定运行一年以上，就可以说接近入门了。

各硬件模块逐个实验理解,逐步实践,再通过自己动手完成一个实际小系统,可以基本掌握底层硬件基础。同时,这个过程,也是软硬件结合学习的基本过程。

(3) 基本掌握底层驱动构件的设计方法。 对本科以上读者,至少掌握 GPIO 构件的设计过程与设计方法(第4章)、UART 构件的设计过程与设计方法(第6章),透彻理解构件化开发方法与底层驱动构件封装规范(第5章)。从而对底层驱动构件有较好的理解与把握。这是一份细致、静心的任务,力戒浮躁,才能理解其要义。书中的底层驱动构件吸取了软件工程的基本原理,学习时注意基本规范。

(4) 掌握单步跟踪调试、打桩调试、printf 输出调试等调试手段。 在初学阶段,充分利用单步跟踪调试了解与硬件打交道的寄存器值的变化,理解 MCU 软件干预硬件的方式。单步跟踪调试也用于底层驱动构件设计阶段。不进入子函数内部执行的单步跟踪调试,可用于整体功能跟踪。打桩调试主要用于编程过程中,功能确认。一般编写几句语句后,即可打桩,调试观察。通过串口 printf 输出信息在 PC 屏幕显示,是嵌入式软件开发中重要的调试跟踪手段①,与 PC 编程中 printf 功能类似,只是嵌入式开发 printf 输出是通过串口输出到 PC 屏幕,PC 上需用串口调试工具显示,PC 编程中 printf 直接将结果显示在 PC 屏幕上。

(5) 日积月累,勤学好问,充分利用本书及相关资源。 有副对联:“智叟何智只顾眼前捞一把,愚公不愚哪管艰苦移二山”。学习嵌入式切忌急功近利,需要日积月累、循序渐进、水滴石穿、十年磨一剑,充分掌握与应用“电子札记”方法。同时,要勤学好问,下真功夫、细功夫。人工智能学科里有个术语叫无教师指导学习模式与有教师指导学习模式,无教师指导学习模式比有教师指导学习模式复杂许多。因此,要多请教良师,少走弯路。此外,本书提供了大量经过打磨的、比较规范的软硬件资源,充分用好这些资源,可以更上一层楼。

以上建议,仅供参考。当然,以上只是基础阶段的学习建议,要成为良好的嵌入式系统设计师,还需要注重理论学习与实践、通用知识与芯片相关知识、硬件知识与软件知识的平衡。要在理解软件工程基本原理的基础上,理解硬件构件与软件构件等基本概念。在实际项目中锻炼,并不断学习与积累经验。

1.3 微控制器与应用处理器简介

1.3.1 微控制器简介

1. 微控制器的基本含义

MCU 是单片微型计算机(单片机)的简称,早期的英文名是 Single-chip Microcomputer,后来大多数称之为微控制器(Microcontroller)或嵌入式计算机(Embedded Computer)。现在 Microcontroller 已经是计算机中一个常用术语,但在1990年之前,大部分英文词典中并没有这个词。我国学者一般使用中文“单片机”一词,而缩写使用“MCU”,

① 本书第6章给出串口 printf 构件,附录C给出其使用方法。

来自于英文"Microcontroller Unit"。所以本书后面的简写一律以 MCU 为准。**MCU 的基本含义是：在一块芯片内集成了中央处理单元（Central Processing Unit，CPU）、存储器（RAM/ROM 等）、定时器/计数器及多种输入输出（I/O）接口的比较完整的数字处理系统。**图 1-1 给出了典型的 MCU 组成框图。

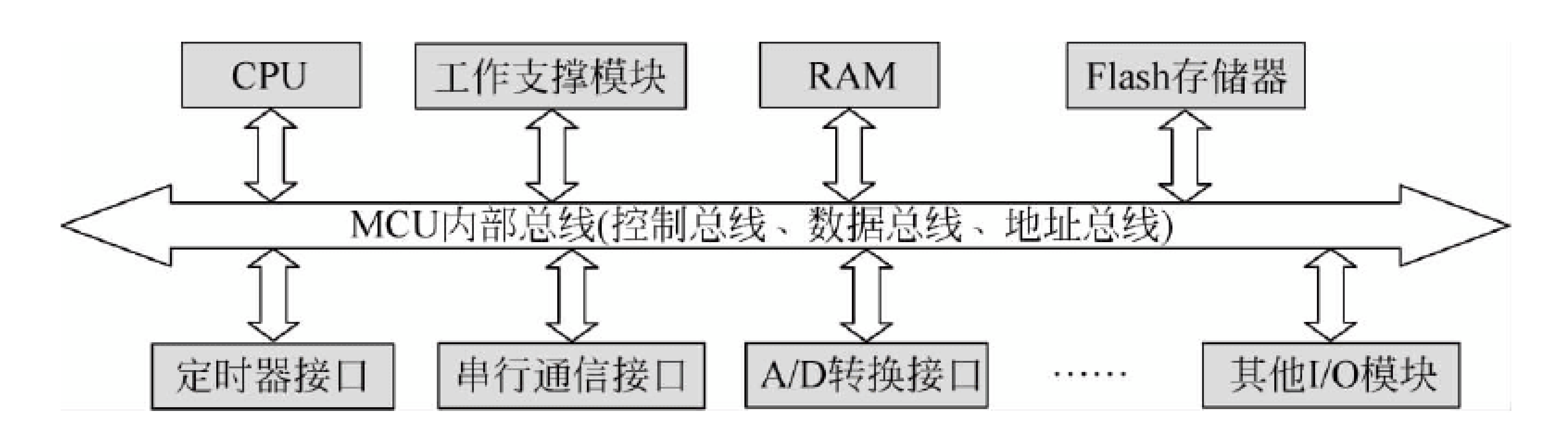


图 1-1 一个典型的 MCU 内部框图

MCU 是在计算机制造技术发展到一定阶段的背景下出现的，它使计算机技术从科学计算领域进入到智能化控制领域。从此，计算机技术在两个重要领域——通用计算机领域和嵌入式（Embedded）计算机领域都获得了极其重要的发展，为计算机的应用开辟了更广阔的空间。

就 MCU 组成而言，虽然它只是一块芯片，但包含计算机的基本组成单元，仍由运算器、控制器、存储器、输入设备、输出设备 5 部分组成，只不过这些都集成在一块芯片内，这种结构使得 MCU 成为具有独特功能的计算机。

2. 嵌入式系统与 MCU 的关系

何立民先生说："有些人搞了十多年的 MCU 应用，不知道 MCU 就是一个最典型的嵌入式系统"①。实际上，MCU 是在通用 CPU 基础上发展起来的，MCU 具有体积小、价格低、稳定可靠等优点，它的出现和迅猛发展，是控制系统领域的一场技术革命。MCU 以其较高的性能价格比、灵活性等特点，在现代控制系统中具有十分重要的地位。**大部分嵌入式系统以 MCU 为核心进行设计。**MCU 从体系结构到指令系统都是按照嵌入式系统的应用特点专门设计的，它能很好地满足应用系统的嵌入、面向测控对象、现场可靠运行等方面的要求。因此**以 MCU 为核心的系统是应用最广的嵌入式系统。**在实际应用时，开发者可以根据具体要求与应用场合，选用最佳型号的 MCU 嵌入到实际应用系统中。

3. MCU 出现之后测控系统设计方法发生的变化

测控系统是现代工业控制的基础，它包含信号检测、处理、传输与控制等基本要素。**在 MCU 出现之前，人们必须用模拟电路、数字电路实现测控系统中的大部分计算与控制功能，这样使得控制系统体积庞大，易出故障。MCU 出现以后，测控系统设计方法逐步产生变化，系统中的大部分计算与控制功能由 MCU 的软件实现。其他电子线路成为 MCU 的外围接口电路，承担着输入、输出与执行动作等功能，而计算、比较与判断等原来必须用电路实现的功能，可以用软件取代，大大地提高了系统的性能与稳定性，这种控制技术称为嵌入式控制技术。在嵌入式控制技术中，核心是 MCU，其他部分依此而展开。下面给出一个典型的以 MCU 为核心的嵌入式测控产品的基本组成。**

① 详见《单片机与嵌入式系统应用》，2004 年第 1 期。

1.3.2　以 MCU 为核心的嵌入式测控产品的基本组成

一个以 MCU 为核心，比较复杂的嵌入式产品或实际嵌入式应用系统，包含模拟量的输入、模拟量的输出、开关量的输入、开关量的输出及数据通信的部分。而所有嵌入式系统中最为典型的则是嵌入式测控系统。图 1-2 给出了一个典型的嵌入式测控系统框图。

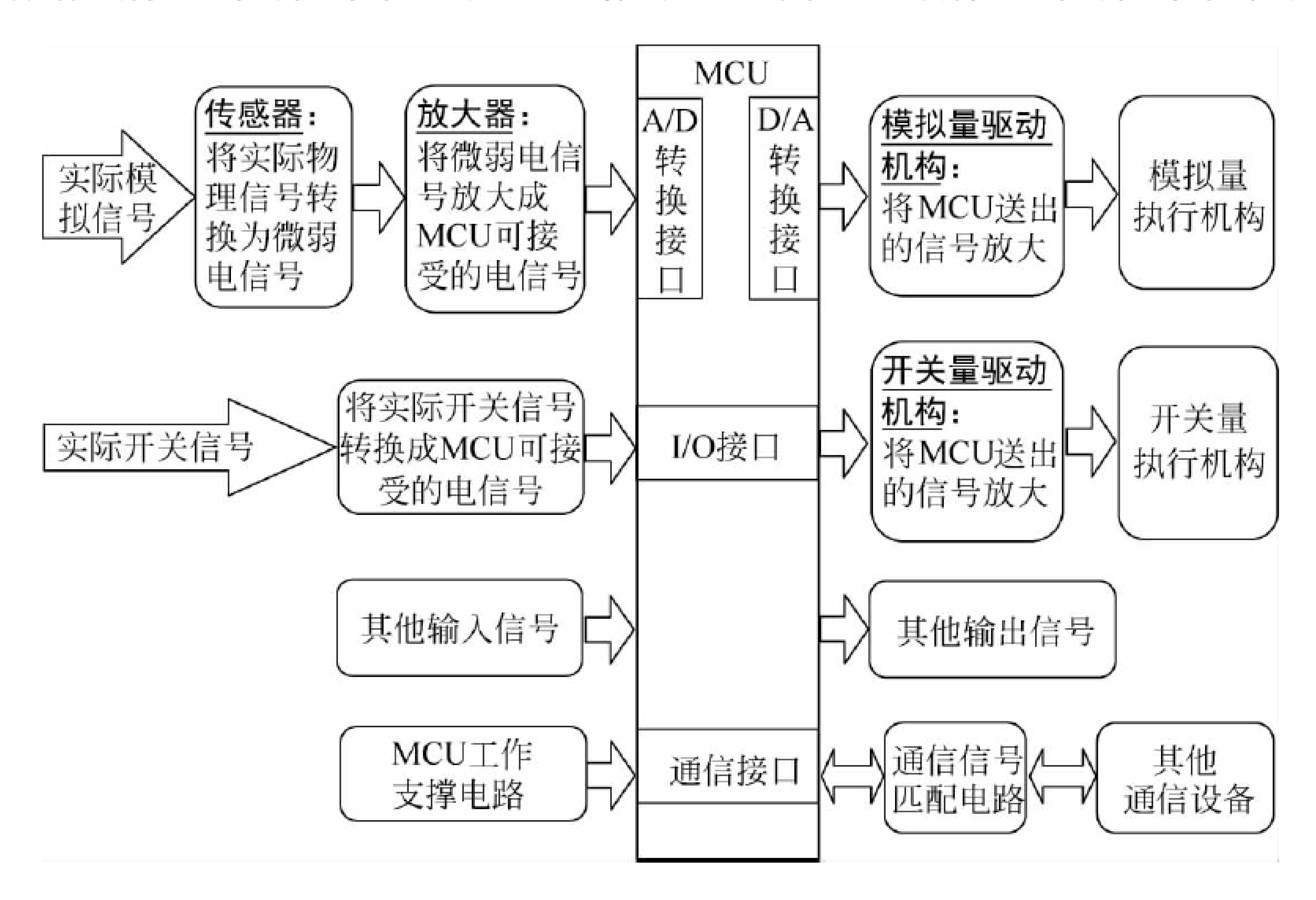


图 1-2　一个典型的嵌入式测控系统框图

1. MCU 工作支撑电路

MCU 工作支撑电路也就是 MCU 硬件最小系统，它保障 MCU 能正常运行，如电源电路、晶振电路及必要的滤波电路等，甚至可包含程序写入器接口电路。

2. 模拟信号输入电路

实际模拟信号一般来自相应的传感器。例如，要测量室内的温度，就需要温度传感器。但是，一般传感器将实际的模拟信号转成的电信号都比较弱，MCU 无法直接获得该信号，需要将其放大，然后经过模/数(AD)转换变为数字信号，进行处理。目前许多 MCU 内部包含 AD 转换模块，实际应用时也可根据需要外接 AD 转换芯片。常见的模拟量有温度、湿度、压力、重量、气体浓度、液体浓度、流量等。对 MCU 来说，模拟信号通过 AD 转换变成相应的数字序列进行处理。

3. 开关量信号输入电路

实际开关信号一般也来自相应的开关类传感器，如光电开关、电磁开关、干簧管(磁开关)、声控开关、红外开关等，一些儿童电子玩具中就有一些类似的开关。手动开关也可作为开关信号送到 MCU 中。对 MCU 来说，开关信号就是只有“0”和“1”两种可能值的数字信号。

4. 其他输入信号或通信电路

其他输入信号通过某些通信方式与 MCU 沟通。常用的通信方式有异步串行(UART)通信、串行外设接口(SPI)通信、并行通信、USB 通信、网络通信等。

5. 输出执行机构电路

在执行机构中，有开关量执行机构，也有模拟量执行机构。开关量执行机构只有“开”“关”两种状态。模拟量执行机构需要连续变化的模拟量控制。MCU 一般不能直接控制这些执行机构，需要通过相应的隔离和驱动电路实现。还有一些执行机构，既不是通常开关量控制，也不是通常 DA 转换量控制，而是“脉冲”量控制，如控制调频电动机，MCU 则通过软件对其控制。

1.3.3　应用处理器简介

1. 应用处理器的基本概念及特点

应用处理器的全名是多媒体应用处理器(Multimedia Application Processor，MAP)。它是在低功耗 CPU 的基础上扩展音视频功能和专用接口的超大规模集成电路。与 MCU 相比，MAP 的最主要特点是：工作频率高；硬件设计更为复杂；软件开发需要选用一个嵌入式操作系统；计算功能更强；抗干扰性能较弱；较少直接应用于控制目标对象；此外，一般情况下，MAP 芯片价格也高于 MCU。

应用处理器是伴随着便携式移动设备特别是智能手机而产生的。手机的技术核心是一个语音压缩芯片，称为基带处理器，发送时对语音进行压缩，接收时解压缩，传输码率只是未压缩的几十分之一，在相同的带宽下可服务更多的人。而智能手机上除通信功能外还增加了数码相机、MP3 播放、FM 广播接收、视频图像播放等功能，基带处理器已经没有能力处理这些新加的功能；另外，视频、音频(高保真音乐)处理的方法和语音不一样，语音只要能听懂，达到传达信息的目的就行了，视频要求亮丽的彩色图像，动听的立体声伴音，目的是使人能得到最大的感官享受。为了实现这些功能，需要另外一个协处理器专门处理这些信号，它就是应用处理器。

针对便携式移动设备，应用处理器的性能需要满足以下几点。

(1) 低功耗，这是因为应用处理器用在便携式移动设备上，通常用电池供电，节能显得格外重要，使用者给电池充满电后希望使用尽可能长的时间。通常 MAP 的核心电压为 0.9～1.2V，接口电压 2.5V 或 3.3V，待机功耗小于 3mW，全速工作时 100～300mW。

(2) 体积微小，因为主要应用在手持式设备中，每一毫米空间都很宝贵。应用处理器通常采用小型 BGA 封装，管脚数有 300～1000 个，锡球直径 0.3～0.6mm，间距 0.45～0.75mm。

(3) 具备尽可能高的性能，目前的便携式移动设备具备了 DAB(Digital Audio Broadcasting)、蓝牙耳机、无线宽带(Wi-Fi)、GPS 导航、3D 游戏等功能，新的功能仍在积极开发中，这些功能都对应用处理器的性能提出了更高的要求。

2. 应用处理器 MAP 与微控制器的接口比较

应用处理器的接口相较于 MCU 更加丰富，除了 MCU 常见的接口，如通用 I/O 即 GPIO、模数转换 AD、数模转换 DA、串行通信接口 UART、串行外设接口 SPI、I2C、CAN、USB、嵌入式以太网、LED、LCD 等之外，因应用处理器的场景多有多媒体、与 PC 方便互连等需要，其接口通常还包括 USB、PCI、TU-R 656、TS、AC97、3D、2D、闪存、DDR、SD 等接口。

3. ARM应用处理器架构及恩智浦应用处理器系列

ARM公司在RISC CPU开发领域中不断取得突破，所设计的微处理器结构从v3发展到v8。2004年之后为避免名称混乱，统一采用Cortex命名，Cortex系列分为M、R、A系列，我们所看到的大部分应用处理器都是基于Cortex-A系列内核的。

Cortex-A系列处理器主要基于32位的ARM v7A或64位的ARM v8A架构。ARM v7A系列支持传统的ARM、Thumb指令集和新增的高性能紧凑型Thumb-2指令集，主要包括了高性能的Cortex-A17和Cortex-A15、可伸缩的Cortex-A9、经过市场验证的Cortex-A8、高效的Cortex-A7和Cortex-A5。ARM v8A是在ARMv7上开发的支持64位数据处理的全新架构，ARMv7架构的主要特性都在ARMv8架构中得到了保留或进一步拓展，该系列主要包括性能最出色、最先进的Cortex-A72、性能优异的Cortex-A57、性能和功耗平衡的Cortex-A53、功耗效率最高的Cortex-A35、体积最小功耗最低的Cortex-A32。

以上仅是应用处理的架构，具体的产品是由应用处理器厂商得到授权后生产的，飞思卡尔(现恩智浦)的i. MX就是其中典型的一个应用处理器系列。

i. MX是基于ARM的单核/多核应用处理器解决方案，适用于汽车电子、工业控制、中高端消费电子、电子书、ePOS、医疗设备、多媒体和显示，以及网络通信等应用，具有可扩展性、性能高和功耗低等特点。i. MX产品主要有2015年推出的i. MX 7系列、2013—2015年间出品的i. MX 6、2010—2012年间出品的i. MX 5X等系列，2009年以前出品的i. MX3x、i. MX2x等。

1.4　嵌入式系统常用术语

在学习嵌入式应用技术的过程中，经常会遇到一些名词术语。从学习规律角度，初步了解这些术语有利于随后的学习。因此，本节对嵌入式系统中所用的一些常用术语给出简要说明，以便读者有个初始印象。

1.4.1　与硬件相关的术语

1. 封装

集成电路的封装(Package)是指用塑料、金属或陶瓷材料等把集成电路封在其中。封装可以保护芯片，并使芯片与外部世界连接。常用的封装形式可分为通孔封装和贴片封装两大类。

通孔封装主要有单列直插(Single-in-line Package，SIP)、双列直插(Dual-in-line Package，DIP)、Z字形直插式封装(Zigzag-in-line Package，ZIP)等。

常见的贴片封装主要有小外形封装(Small Outline Package，SOP)、紧缩小外形封装(Shrink Small Outline Package，SSOP)、四方扁平封装(Quad-Flat Package，QFP)、塑料薄方封装(Plastic-Low-profile Quad-Flat Package，LQFP)、塑料扁平组件式封装(Plastic Flat Package，PFP)、插针网格阵列封装(Ceramic Pin Grid Array Package，PGA)、球栅阵列封装(Ball Grid Array Package，BGA)等。

2. 印刷电路板

印刷电路板(Printed Circuit Board,PCB)是组装电子元件用的基板,是在通用基材上按预定设计形成点间连接及印制元件的印制板,是电路原理图的实物化。PCB的主要功能是提供集成电路等各种电子元器件固定、装配的机械支撑;实现集成电路等各种电子元器件之间的布线和电气连接(信号传输)或电绝缘;为自动装配提供阻焊图形,为元器件插装、检查、维修提供识别字符和图形等。

3. 动态可读写随机存储器与静态可读写随机存储器

动态可读写随机存储器(Dynamic Random Access Memory,DRAM),由一个MOS管组成一个二进制存储位。MOS管的放电导致表示“1”的电压会慢慢降低。一般每隔一段时间就要控制刷新信息,给其充电。DRAM价格低,但控制烦琐,接口复杂。

静态可读写随机存储器(Static Random Access Memory,SRAM),一般由4个或者6个MOS管构成一个二进制位。当电源有电时,SRAM不用刷新,可以保持原有的数据。

4. 只读存储器

只读存储器(Read Only Memory,ROM),数据可以读出,但不可以修改,所以称为只读存储器。通常存储一些固定不变的信息,如常数、数据、换码表、程序等。它具有断电后数据不丢失的特点。ROM有固定ROM、可编程ROM(即PROM)和可擦除ROM(即EPROM)三种。

PROM的编程原理是通过大电流将相应位的熔丝熔断,从而将该位改写成0,熔丝熔断后不能再次改变,所以只改写一次。

EPROM(Erase PROM)是可以擦除和改写的ROM,它用MOS管代替了熔丝,所以可以反复擦除、多次改写。擦除是用紫外线擦除器来完成的,很不方便。有一种用低电压信号即可擦除的EPROM称为电可擦除EPROM,简写为E^2PROM或EEPROM(Electrically Erasable Programmable Read-Only Memory)。

5. 闪速存储器

闪速存储器(Flash Memory)简称闪存,是一种新型快速的E^2PROM。由于工艺和结构上的改进,闪存比普通的E^2PROM的擦除速度更快,集成度更高。闪存相对于传统的E^2PROM来说,其最大的优点是系统内编程,也就是说不需要另外的器件来修改内容。闪存的结构随着时代的发展而有些变动,尽管现代的快速闪存是系统内可编程的,但仍然没有RAM使用起来方便。擦写操作必须通过特定的程序算法来实现。

6. 模拟量与开关量

模拟量是指时间连续、数值也连续的物理量,如温度、压力、流量、速度、声音等。在工程技术上,为了便于分析,常用传感器、变换器将模拟量转换为电流、电压或电阻等电学量。

开关量是指一种二值信号,用两个电平(高电平和低电平)分别来表示两个逻辑值(逻辑1和逻辑0)。

1.4.2 与通信相关的术语

1. 并行通信

并行通信是指数据的各位同时在多根并行数据线上进行传输的通信方式,数据的各位同时由源到达目的地。适合近距离、高速通信。常用的有4位、8位、16位、32位等同时

传输。

2. 串行通信

串行通信是指数据在单线(电平高低表征信号)或双线(差分信号)上,按时间先后一位一位地传送,其优点是节省传输线,但相对于并行通信来说,速度较慢。在嵌入式系统中,串行通信一词一般特指用串行通信接口 UART 与 RS232 芯片连接的通信方式。下面介绍的 SPI、I2C、USB 等通信方式也属于串行通信,但由于历史发展和应用领域的不同,它们分别使用不同的专用名词来命名。

3. 串行外设接口

串行外设接口(Serial Peripheral Interface,SPI)也是一种串行通信方式,主要用于 MCU 扩展外围芯片使用。这些芯片可以是具有 SPI 接口的 AD 转换、时钟芯片等。

4. 集成电路互连总线

集成电路互连总线(Inter-Integrated Circuit,I2C)是一种由 PHILIPS 公司开发的两线式串行总线,有的书籍也记为 IIC 或 I^2C,主要用于用户电路板内 MCU 与其外围电路的连接。

5. 通用串行总线

通用串行总线(Universal Serial Bus,USB)是 MCU 与外界进行数据通信的一种新的方式,其速度快,抗干扰能力强,在嵌入式系统中得到了广泛的应用。USB 不仅成为通用计算机上最重要的通信接口,也是手机、家电等嵌入式产品的重要通信接口。

6. 控制器局域网

控制器局域网(Controller Area Network,CAN)是一种全数字、全开放的现场总线控制网络,目前在汽车电子中应用最广。

7. 背景调试模式

背景调试模式(Background Debug Mode,BDM)是 Freescale 半导体公司提出的一种调试接口,主要用于嵌入式 MCU 的程序下载与程序调试。

8. 边界扫描测试协议

边界扫描测试协议(Joint Test Action Group,JTAG)是由国际联合测试行动组开发,对芯片进行测试的一种方式,可将其用于对 MCU 的程序进行载入与调试。JTAG 能获取芯片寄存器等内容,或者测试遵守 IEEE 规范的器件之间引脚连接情况。

9. 串行线调试技术

串行线调试(Serial Wire Debug,SWD)技术使用 2 针调试端口,是 JTAG 的低针数和高性能替代产品,通常用于小封装微控制器的程序写入与调试。SWD 适用于所有 ARM 处理器,兼容 JTAG。

关于通信相关的术语还有嵌入式以太网、无线传感器网络、ZigBee、射频通信等,本章不再进一步介绍。

1.4.3 与功能模块相关的术语

1. 通用输入/输出

通用输入/输出(General Purpose I/O,GPIO),即基本的输入/输出,有时也称并行 I/O。作为通用输入引脚时,MCU 内部程序可以读取该引脚,知道该引脚是“1”(高电平)或“0”

(低电平),即开关量输入。作为通用输出引脚时,MCU 内部程序向该引脚输出"1"(高电平)或"0"(低电平),即开关量输出。

2. 模数转换与数模转换

模数转换(ADC)的功能是将电压信号(模拟量)转换为对应的数字量。实际应用中,这个电压信号可能由温度、湿度、压力等实际物理量经过传感器和相应的变换电路转化而来。经过 AD 转换,MCU 就可以处理这些物理量。而与之相反,数模转换(DAC)的功能则是将数字量转换为电压信号(模拟量)。

3. 脉冲宽度调制器

脉冲宽度调制器(Pulse Width Modulator,PWM),是一个 DA 转换器,可以产生一个高电平和低电平之间重复交替的输出信号,这个信号就是 PWM 信号。

4. 看门狗

看门狗(Watch Dog),是一种为了防止程序跑飞而设计的自动定时器。当程序跑飞时,由于无法正常执行清除看门狗定时器,看门狗定时器会自动溢出,使系统程序复位。

5. 液晶显示

液晶显示(Liquid Crystal Dispaly,LCD),是电子信息产品的一种显示器件,可分为字段型、点阵字符型、点阵图形型三类。

6. 发光二极管

发光二极管(Light Emitting Diode,LED),是一种将电流顺向通到半导体 PN 结处而发光的器件。常用于家电指示灯、汽车灯和交通警示灯。

7. 键盘

键盘是嵌入式系统中最常见的输入设备。识别键盘是否有效被按下的方法有查询法、定时扫描法与中断法等。

与功能模块相关的术语很多,本章不再进一步介绍,具体学习时逐步积累。

1.5　嵌入式系统常用的 C 语言基本语法概要

本书主要使用 C 语言阐述嵌入式技术基础。

C 语言是在 20 世纪 70 年代初问世的。1978 年,美国电话电报公司(AT&T)贝尔实验室正式发表了 C 语言。由 B. W. Kernighan 和 D. M. Ritchit 合著的 *THE C PROGRAMMING LANGUAGE* 一书,被简称为 K&R,也有人称之为 K&R 标准。但是,在 K&R 中并没有定义一个完整的标准 C 语言,后来由美国国家标准学会在此基础上制定了一个 C 语言标准,于 1983 年发表,通常称之为 ANSI C 或标准 C。

本节简要介绍 C 语言的基本知识,特别是和嵌入式系统编程密切相关的基本知识,未学过标准 C 语言的读者可以通过本节了解 C 语言,以后通过实例逐步积累相关编程知识。对 C 语言很熟悉的读者,可以跳过本节。

1.5.1　C 语言的运算符与数据类型

1. C 语言的运算符

C 语言的运算符分为算术、逻辑、关系和位运算及一些特殊的操作符。表 1-1 列出了 C

语言的常用运算符及使用方法举例。

表 1-1　C 语言的常用运算符

运算类型	运算符	简明含义
算术运算	+、-、*、/、%	加、减、乘、除、取模
逻辑运算	‖、&&、!	逻辑或、逻辑与、逻辑非
关系运算	>、<、>=、<=、==、!=	大于、小于、大于等于、小于等于、等于、不等于
位运算	~、<<、>>、&、^、|	按位取反、左移、右移、按位与、按位异或、按位或
增量和减量	++、--	增量运算符、减量运算符
复合赋值	+=、-=、>>=、<<=	加法赋值、减法赋值、右移位赋值、左移位赋值
	*=、|=、&=、^=	乘法赋值、按位或赋值、按位与赋值、按位异或赋值
	%=、/=	取模赋值、除法赋值
指针和地址	*、&	取内容、取地址
输出格式转换	0x、0o、0b、0u	无符号十六、八、二、十进制数
	0d	带符号十进制数

2. C 语言的数据类型

C 语言的数据类型有基本数据类型和构造数据类型两大类。**基本数据类型是指字节型、整型及实型**，如表 1-2 所示。

表 1-2　C 语言基本数据类型

数据类型		简明含义	位数	字节数	值域
字节型	signed char	有符号字节型	8	1	-128～+127
	unsigned char	无符号字节型	8	1	0～255
整型	signed short	有符号短整型	16	2	-32 768～+32 767
	unsigned short	无符号短整型	16	2	0～65 535
	signed int	有符号整型	16	2	-32 768～+32 767
	unsigned int	无符号整型	16	2	0～65 535
	signed long	有符号长整型	32	4	-2 147 483 648～+2 147 483 647
	unsigned long	无符号长整型	32	4	0～4 294 967 295
实型	float	浮点型	32	4	约 $\pm 3.4\times(10^{-38}\sim10^{+38})$
	double	双精度型	64	8	约 $\pm 1.7\times(10^{-308}\sim10^{+308})$

构造数据类型有数组、指针、枚举、结构体、共用体和空类型。枚举是一个被命名为整型常量的集合。结构体和共用体是基本数据类型的组合。空类型字节长度为 0，主要有两个用途：一是明确地表示一个函数不返回任何值；二是产生一个同一类型指针(可根据需要动态地分配给其内存)。

嵌入式中还常用到寄存器类型(Register)变量，简要说明如下。通常，内存变量(包括全局变量、静态变量、局部变量)的值存放在内存中。CPU 访问内存变量要通过三总线(地址总线、数据总线、控制总线)进行，如果有一些变量使用频繁，则为存取变量的值要花不少时间。为提高执行效率，C 语言允许使用关键字“register”声明，将少量局部变量的值放在 CPU 中内部寄存器中，需要用时直接从寄存器取出参加运算，不必再到内存中存取。关于 register 类型变量的使用需注意：①只有局部变量和形式参数可以使用寄存器变量，其他变

量(如全局变量、静态变量)不能使用 register 类型变量；②CPU 内部寄存器数目很少，不能定义任意多个寄存器变量。

1.5.2　程序流程控制

在程序设计中主要有三种基本控制结构：顺序结构、选择结构和循环结构。

1. 顺序结构

顺序结构就是从前向后依次执行语句。从整体上看，所有程序的基本结构都是顺序结构，中间的某个过程可以是选择结构或循环结构。

2. 选择结构

在大多数程序中都会包含选择结构。其作用是，根据所指定的条件是否满足，决定执行哪些语句。在 C 语言中主要有 if 和 switch 两种选择结构。

1）if 结构

```
if (表达式)语句项;
```

或

```
if (表达式)
    语句项;
else
    语句项;
```

如果表达式取值真(除 0 以外的任何值)，则执行 if 的语句项；否则，如果 else 存在的话，就执行 else 的语句项。每次只会执行 if 或 else 中的某一个分支。语句项可以是单独的一条语句，也可以是多条语句组成的语句块(要用一对大括号“{}”括起来)。

if 语句可以嵌套，有多个 if 语句时 else 与最近的一个配对。对于多分支语句，可以使用 if…else if…else if…else…的多重判断结构，也可以使用下面讲到的 switch 开关语句。

2）switch 结构

switch 是 C 语言内部多分支选择语句，它根据某些整型和字符常量对一个表达式进行连续测试，当一常量值与其匹配时，它就执行与该变量有关的一个或多个语句。switch 语句的一般形式如下。

```
switch(表达式)
{
    case 常数 1:
        语句项 1;
        break;
    case 常数 2:
        语句项 2;
        break;
    …
    default:
        语句项;
}
```

根据case语句中所给出的常量值,按顺序对表达式的值进行测试,当常量与表达式值相等时,就执行这个常量所在的case后的语句块,直到碰到break语句,或者switch的末尾为止。若没有一个常量与表达式值相符,则执行default后的语句块。default是可选的,如果它不存在,并且所有的常量与表达式值都不相符,那就不做任何处理。

switch语句与if语句的不同之处在于switch只能对等式进行测试,而if可以计算关系表达式或逻辑表达式。

break语句在switch语句中是可选的,但是不用break,则从当前满足条件的case语句开始连续执行后续指令,不判断后续case语句的条件,一直到碰到break或switch的末尾为止。为了避免输出不应有的结果,在每一case语句之后加break语句,使每一次执行之后均可跳出switch语句。

3. 循环结构

C语言中的循环结构常用for循环、while循环与do…while循环。

1) for循环

格式为:

```
for(初始化表达式; 条件表达式; 修正表达式)
    {循环体}
```

执行过程为:先求解初始化表达式;再判断条件表达式,若为假(0),则结束循环,转到循环下面的语句;如果其值为真(非0),则执行“循环体”中语句。然后求解修正表达式;再转到判断条件表达式处根据情况决定是否继续执行“循环体”。

2) while循环

格式为:

```
while(条件表达式)
    {循环体}
```

当表达式的值为真(非0)时执行循环体。其特点是:先判断后执行。

3) do…while循环

格式为:

```
do
    {循环体}
while(条件表达式);
```

其特点是:先执行后判断。即当流程到达do后,立即执行循环体一次,然后才对条件表达式进行计算、判断。若条件表达式的值为真(非0),则重复执行一次循环体。

4. break和continue语句在循环中的应用

在循环中常常使用break语句和continue语句,这两个语句都会改变循环的执行情况。break语句用来从循环体中强行跳出循环,终止整个循环的执行;continue语句使其后语句不再被执行,进行新的一次循环(可以形象地理解为返回循环开始处执行)。

1.5.3　函数

所谓函数，即子程序，也就是“语句的集合”，就是说把经常使用的语句群定义成函数，供其他程序调用，**函数的编写与使用要遵循软件工程的基本规范**。

使用函数要注意：函数定义时要同时声明其类型；调用函数前要先声明该函数；传给函数的参数值，其类型要与函数原定义一致；接收函数返回值的变量，其类型也要与函数类型一致等。**函数传参有传值与传址之分**。

函数的返回值：return 表达式；

return 语句用来立即结束函数，并返回一确定值给调用程序。如果函数的类型和 return 语句中表达式的值不一致，则以函数类型为准。对数值型数据，可以自动进行类型转换。即函数类型决定返回值的类型。

1.5.4　数据存储方式

C 语言中，存储与操作方式除基本变量方式外，还有数组、指针、结构体、共用体，简介如下。此外，数据类型还可使用 typedef 定义别名，方便使用。

1. 数组

在 C 语言中，数组是一个构造类型的数据，是由基本类型数据按照一定的规则组成的。构造类型还包括结构体类型、共用体类型。数组是有序数据的集合，数组中的每一个元素都属于同一个数据类型。用一个统一的数组名和下标唯一地确定数组中的元素。

1）一维数组的定义和引用

定义方式为：类型说明符 数组名[常量表达式]；

其中，数组名的命名规则和变量相同。定义数组的时候，需要指定数组中元素的个数，即常量表达式需要明确设定，不可以包含变量。例如：

```
int a[10];        //定义了一个整型数组,数组名为 a,有 10 个元素,下标 0～9
```

数组必须先定义，然后才能使用。而且只能通过下标一个一个地访问。形如：数组名[下标]。

2）二维数组的定义和引用

定义方式为：类型说明符 数组名[常量表达式][常量表达式]

例如：

```
float  a[3][4];     //定义 3 行 4 列的数组 a,下标 0～2,0～3
```

其实，二维数组可以看成是两个一维数组。可以把 a 看作是一个一维数组，它有三个元素：a[0]，a[1]，a[2]，而每个元素又是一个包含 4 个元素的一维数组。二维数组的表示形式为：数组名[下标][下标]。

3）字符数组

用于存放字符数据(char 类型)的数组是字符数组。字符数组中的一个元素存放一个字符。例如：

```
char c[5];
c[0] = 't';c[1] = 'a'; c[2] = 'b'; c[3] = 'l'; c[4] = 'e';
//字符数组 c[5]中存放的就是字符串"table"。
```

在C语言中，是将字符串作为字符数组来处理的。但是，在实际应用中，关于字符串的实际长度，C语言规定了一个“字符串结束标志”，以字符'\0'作为标志(实际值0x00)。即如果有一个字符串，前面n－1个字符都不是空字符(即'\0')，而第n个字符是'\0'，则此字符的有效字符为n－1个。

4）动态数组

动态数组是相对于静态数组而言。静态数组的长度是预先定义好的，在整个程序中，一旦给定大小后就无法改变。而动态数组则不然，它可以随程序需要而重新指定大小。动态数组的内存空间是从堆上分配(即动态分配)的，是通过执行代码而为其分配存储空间。当程序执行到这些语句时，才为其分配。程序员自己负责释放内存。

在C语言中，可以通过malloc、calloc函数，进行内存空间的动态分配，从而实现数组的动态化，以满足实际需求。

5）数组如何模拟指针的效果

其实，数组名就是一个地址，一个指向这个数组元素集合的首地址。可以通过数组加位置的方式进行数组元素的引用。例如：

```
int a[5];     //定义了一个整型数组,数组名为 a,有 5 个元素,下标 0～4
```

访问到数组a的第三个元素方式有：方式一：a[2]；方式二：*(a+2)，关键是数组的名称本身就可以当作地址看待。

2. 指针

指针是C语言中广泛使用的一种数据类型，运用指针是C语言最主要的风格之一。在嵌入式编程中，指针尤为重要。利用指针变量可以表示各种数据结构，很方便地使用数组和字符串，并能像汇编语言一样处理内存地址，从而编出精练而高效的程序。但是使用指针要特别细心，计算得当，避免指向不适当区域。

指针是一种特殊的数据类型，在其他语言中一般没有。指针是指向变量的地址，实质上指针就是存储单元的地址。根据所指的变量类型不同，可以是整型指针(int *)、浮点型指针(float *)、字符型指针(char *)、结构指针(struct *)和联合指针(union *)。

1）指针变量的定义

其一般形式为：类型说明符　*变量名；

其中，*表示这是一个指针变量，变量名即为定义的指针变量名，类型说明符表示本指针变量所指向的变量的数据类型。例如：

```
int *p1;      //表示 p1 是指向整型数的指针变量,p1 的值是整型变量的地址
```

2）指针变量的赋值

指针变量同普通变量一样，使用之前不仅要进行声明，而且必须赋予具体的值。未经赋值的指针变量不能使用，否则将造成系统混乱，甚至死机。指针变量的赋值只能赋予地址。

例如：

```
int a;                  //a 为整型数据变量
int *p1;                //声明 p1 是整型指针变量
p1 =&a;                 //将 a 的地址作为 p1 初值
```

3）指针的运算

（1）取地址运算符 &：取地址运算符 & 是单目运算符，其结合性为自右至左，其功能是取变量的地址。

（2）取内容运算符 *：取内容运算符 * 是单目运算符，其结合性为自右至左，用来表示指针变量所指的变量。在 * 运算符之后跟的变量必须是指针变量。例如：

```
int a,b;                //a,b 为整型数据变量
int *p1;                //声明 p1 是整型指针变量
p1 =&a;                 //将 a 的地址作为 p1 初值
a=80;
b=*p1;                  //运行结果:b=80,即为 a 的值
```

注意：取内容运算符"*"和指针变量声明中的"*"虽然符号相同，但含义不同。在指针变量声明中，"*"是类型说明符，表示其后的变量是指针类型。而表达式中出现的"*"则是一个运算符用以表示指针变量所指的变量。

（3）指针的加减算术运算：对于指向数组的指针变量，可以加/减一个整数 n（由于指针变量实质是地址，给地址加/减一个非整数就错了）。设 pa 是指向数组 a 的指针变量，则 pa＋n，pa－n，pa＋＋，＋＋pa，pa－－，－－pa 运算都是合法的。指针变量加/减一个整数 n 的意义是把指针指向的当前位置（指向某数组元素）向前或向后移动 n 个位置。

注意：数组指针变量前/后移动一个位置和地址加/减 1 在概念上是不同的。因为数组可以有不同的类型，各种类型的数组元素所占的字节长度是不同的。如指针变量加 1，即向后移动一个位置，表示指针变量指向下一个数据元素的首地址，而不是在原地址基础上加 1。例如：

```
int a[5],*pa;           //声明 a 为整型数组(下标为 0～4),pa 为整型指针
pa=a;                   //pa 指向数组 a,也是指向 a[0]
pa=pa+2;                //pa 指向 a[2],即 pa 的值为 &pa[2]
```

注意：指针变量的加/减运算只能对数组指针变量进行，对指向其他类型变量的指针变量作加/减运算是毫无意义的。

4）void 指针类型

顾名思义，void * 为"无类型指针"，即用来定义指针变量，不指定它是指向哪种类型数据，但可以把它强制转化成任何类型的指针。

众所周知，如果指针 p1 和 p2 的类型相同，那么可以直接在 p1 和 p2 间互相赋值；如果 p1 和 p2 指向不同的数据类型，则必须使用强制类型转换运算符把赋值运算符右边的指针类型转换为左边指针的类型。例如：

```
float *p1;              //声明 p1 为浮点型指针
int *p2;                //声明 p2 为整型指针
p1 = (float *)p2;       //强制转换整型指针 p2 为浮点型指针值给 p1 赋值
```

而 void * 则不同,任何类型的指针都可以直接赋值给它,无须进行强制类型转换。

```
void *p1;             //声明 p1 无类型指针
int *p2;              //声明 p2 为整型指针
p1 = p2;              //用整型指针 p2 的值给 p1 直接赋值
```

但这并不意味着,"void *"也可以无须强制类型转换地赋给其他类型的指针,也就是说 p2=p1 这条语句编译就会出错,而必须将 p1 强制类型转换成"int *"类型。因为"无类型"可以包容"有类型",而"有类型"则不能包容"无类型"。

3. 结构体

结构体是由基本数据类型构成的,并用一个标识符来命名的各种变量的组合。结构体中可以使用不同的数据类型。

1) 结构体的说明和结构体变量的定义

例如,定义一个名为 student 的结构体变量类型:

```
struct student                  //定义一个名为 student 的结构体变量类型
{
    char name[8];               //成员变量"name"为字符型数组
    char class[10];             //成员变量"class"为字符型数组
    int age;                    //成员变量"age"为整型
};
```

这样,若声明 s1 为一个 student 类型的结构体变量,则使用如下语句:

```
struct student s1;          //声明 s1 为 student 类型的结构体变量
```

又例如,定义一个名为 student 的结构体变量类型,同时声明 s1 为一个 student 类型的结构体变量:

```
Struct student              //定义一个名为 student 的结构体变量类型
{
    char name[8];           //成员变量"name"为字符型数组
    char class[10];         //成员变量"class"为字符型数组
    int age;                //成员变量"age"为整型
}s1;                        //声明 s1 为 student 类型的结构体变量
```

2) 结构体变量的使用

结构体是一个新的数据类型,因此结构体变量也可以像其他类型的变量一样赋值运算,不同的是结构体变量以成员作为基本变量。

结构体成员的表示方式为：

```
结构体变量.成员名
```

如果将“**结构体变量.成员名**”看成一个整体，则这个整体的数据类型与结构体中该成员的数据类型相同，这样就像前面所讲的变量那样使用。例如：

```
s1.age=18;        //将数据 18 赋给 s1.age(理解为学生 s1 的年龄为 18)
```

3）结构体指针

结构体指针是指向结构体的指针。它由一个加在结构体变量名前的“*”操作符来声明。例如，用上面已说明的结构体声明一个结构体指针如下：

```
struct student *Pstudent;     //声明 Pstudent 为一个 student 类型指针
```

使用结构体指针对结构体成员的访问，与结构体变量对结构体成员的访问在表达方式上有所不同。结构体指针对结构体成员的访问表示为：

```
结构体指针名->结构体成员
```

其中，“->”是两个符号“-”和“>”的组合，好像一个箭头指向结构体成员。例如，要给上面定义的结构体中 name 和 age 赋值，可以用下面的语句：

```
strcpy(Pstudent->name,"LiuYuZhang");
Pstudent->age=18;
```

实际上，Pstudent->name 就是(*Pstudent).name 的缩写形式。

需要指出的是**结构体指针是指向结构体的一个指针，即结构体中第一个成员的首地址，**因此在使用之前应该对结构体指针初始化，即分配整个结构体长度的字节空间。这可用下面的函数完成：

```
Pstudent=(struct student *)malloc(sizeof(struct student));
```

sizeof(struct student)自动求取 student 结构体的字节长度，malloc()函数定义了一个大小为结构体长度的内存区域，然后将其地址作为结构体指针返回。

4. 共用体

在进行某些算法的 C 语言编程时，需要使几种不同类型的变量之间切换，可以将它们存放到同一段内存单元中。也就是使用覆盖技术，几个变量互相覆盖。这种几个不同的变量共同占用一段内存的结构，在 C 语言中被称作“共用体”类型结构，简称共用体。语法如下：

```
union 共用体名
    {
    成员表列
    }变量表列;
```

有的文献中文翻译为“联合体”,似乎不妥,中文使用“共用体”一词更为妥当。

5. 用 typedef 定义类型

除了可以直接使用C提供的标准类型名(如 int、char、float、double、long 等)和自己定义的结构体、指针、枚举等类型外,还可以用 typedef 定义新的类型名来代替已有的类型名。例如:

```
typedef unsigned char uint_8;
```

指定用 uint_8 代表 unsigned char 类型。这样下面的两个语句是等价的:

```
unsigned char n1;
```

等价于

```
uint_8 n1;
```

用法说明:

(1) 用 typedef 可以定义各种类型名,但不能用来定义变量。

(2) 用 typedef 只是对已经存在的类型增加一个类型别名,而没有创造新的类型。

(3) typedef 与 #define 有相似之处,如:

```
typedef unsigned int uint_16;
#define uint_16 unsigned int;
```

这两句的作用都是用 uint_16 代表 unsigned int(注意顺序)。但事实上它们二者不同,#define 是在预编译时处理,它只能做简单的字符串替代,而 typedef 是在编译时处理。

(4) 当不同源文件中用到各种类型数据(尤其是像数组、指针、结构体、共用体等较复杂数据类型)时,常用 typedef 定义一些数据类型,并把它们单独存放在一个文件中,然后在需要用到它们时,用 #include 命令把该文件包含进来。

(5) 使用 typedef 有利于程序的通用与移植。特别是用 typedef 定义结构体类型,在嵌入式程序中常用到。例如:

```
typedef struct student
{
    char name[8];
    char class[10];
    int age;
}STU;
```

以上声明了新类型名 STU,代表一个结构体类型。可以用该新的类型名来定义结构体变量。例如:

```
STU student1;          //定义 STU 类型的结构体变量 student1
STU *S1;               //定义 STU 类型的结构体指针变量 *S1
```

1.5.5　编译预处理

C 语言提供编译预处理的功能,“编译预处理”是 C 编译系统的一个重要组成部分。C

语言允许在程序中使用几种特殊的命令(它们不是一般的 C 语句)。在 C 编译系统对程序进行通常的编译(包括语法分析、代码生成、优化等)之前,先对程序中的这些特殊的命令进行“预处理”,然后将预处理的结果和源程序一起再进行常规的编译处理,以得到目标代码。C 提供的预处理功能主要有宏定义、条件编译和文件包含。

1. 宏定义

```
#define 宏名 表达式
```

表达式可以是数字、字符,也可以是若干条语句。在编译时,所有引用该宏的地方,都将自动被替换成宏所代表的表达式。例如:

```
#define PI 3.1415926          //以后程序中用到数字 3.1415926 就写 PI
#define S(r) PI*r*r           //以后程序中用到 PI*r*r 就写 S(r)
```

2. 撤销宏定义

```
#undef 宏名
```

3. 条件编译

```
#if 表达式
#else 表达式
#endif
```

如果表达式成立,则编译#if 下的程序,否则编译#else 下的程序,#endif 为条件编译的结束标志。

```
#ifdef 宏名        //如果宏名称被定义过,则编译以下程序
#ifndef 宏名       //如果宏名称未被定义过,则编译以下程序
```

条件编译通常用来调试、保留程序(但不编译),或者在需要对两种状况做不同处理时使用。

4. “文件包含”处理

所谓“文件包含”是指一个源文件将另一个源文件的全部内容包含进来,其一般形式为:

```
#include "文件名"
```

小　　结

本章给出嵌入式系统基本概念、由来、发展简史、分类及特点;给出嵌入式系统的学习困惑、知识体系与学习建议;给出微控制器 MCU 及应用处理器 MAP 的简介;简要归纳了

嵌入式系统的常用术语及C语言基本语法。

(1) 关于嵌入式系统定义：可以表述为嵌入式系统是一种计算机硬件和软件的组合，也许还有机械装置，用于实现一个特定功能。在某些特定情况下，嵌入式系统是一个大系统或产品的一部分。从计算机本身角度可将嵌入式系统概括表述为：嵌入式系统，即嵌入式计算机系统，它是不以计算机面目出现的"计算机"，这个计算机系统隐含在各类具体的产品之中，这些产品中，计算机程序起到了重要作用。关于嵌入式系统的由来，可以表述为：计算机是因科学家需要一个高速的计算工具而产生的，而嵌入式计算机系统测控系统对计算机需要而逐步产生的。关于嵌入式系统分类，可以按应用范围简单地把嵌入式系统分为电子系统智能化(微控制器类)和计算机应用延伸(应用处理器)这两大类。关于嵌入式系统特点，可以从与通用计算机比较的角度，表述为嵌入式系统是不单独以通用计算机的面目出现的计算机系统，它的开发需要专用工具和特殊方法，使用MCU设计嵌入式系统，数据与程序空间采用不同存储介质，开发嵌入式系统涉及软件、硬件及应用领域的知识等。

(2) 分析了一些初学者在学习嵌入式系统时可能遇到的困惑，如：选择入门芯片：是微控制器还是应用处理器？开始学习阶段时，是无操作系统(NOS)、实时操作系统(RTOS)，还是一般嵌入式操作系统(EOS)？硬件与软件如何平衡？本书的建议是：使用微控制器而不是使用应用处理器作为入门芯片；开始阶段，不学习操作系统，着重打好底层驱动的使用方法、设计方法等软硬件基础。关于硬件与软件平衡问题，可以表述为嵌入式系统与硬件紧密相关，是软件与硬件的综合体，没有对硬件的理解就不可能写好嵌入式软件，同样没有对软件的理解也不可能设计好嵌入式硬件。关于嵌入式系统的知识体系可以简单表述为：芯片最小硬件系统及软件最小系统，各种模块的底层驱动构件使用方法及构件的设计方法，掌握在驱动构件基础上遵循软件工程原则的应用软件的开发方法，掌握嵌入式基本调试方法等。给出的学习建议主要有：遵循"先易后难，由浅入深"的原则，打好软硬件基础；充分理解知识要素、掌握底层驱动构件的使用方法；基本掌握底层驱动构件的设计方法；掌握单步跟踪调试、打桩调试、printf输出调试等调试手段；日积月累，勤学好问，充分利用本书及相关资源。关键点是学习嵌入式切忌急功近利，需要日积月累、循序渐进、水滴石穿、十年磨一剑。

(3) MCU的基本含义是：在一块芯片内集成了CPU、存储器、定时器/计数器及多种输入输出(I/O)接口的比较完整的数字处理系统。以MCU为核心的系统是应用最广的嵌入式系统，是现代测控系统的核心。MCU出现之前，人们必须用纯硬件电路实现测控系统，MCU出现以后，测控系统中的大部分计算与控制功能由MCU的软件实现，输入、输出与执行动作等通过硬件实现，带来了设计上的本质变化。应用处理器的全名是多媒体应用处理器，简称MAP。它是在低功耗CPU的基础上扩展音视频功能和专用接口的超大规模集成电路，其功能与开发方法接近PC。

(4) 简要归纳了嵌入式系统的硬件、通信、功能模块等方面的术语，目的是对嵌入式系统基本词汇有初步认识，为后续各章学习提供基础。

(5) 简要给出嵌入式系统常用的C语言基本语法概要，目的是快速收拢与复习本书所用到的C语言基本知识要素。

习　　题

1. 简要总结嵌入式系统定义、由来、分类及特点。

2. 用 450 字以上，500 字以内，概括你对 ARM 的认识。

3. 归纳嵌入式系统的学习困惑，简要说明如何消除这些困惑。

4. 简要归纳嵌入式系统的知识体系。

5. 结合书中给出的嵌入式系统基础阶段的学习建议，从个人的角度，你认为应该如何学习嵌入式系统？

6. 简要给出 MCU 的定义及典型内部框图。

7. 举例给出一个具体的以 MCU 为核心的嵌入式测控产品的基本组成。

8. 简要比较中央处理器 CPU、微控制器 MCU 与应用处理器 MAP。

9. 列表罗列嵌入式系统常用术语（中文名、英文缩写、英文全写）。

10. 说明全局变量、局部变量、常数、程序机器码的存储特征。

11. 比较 C 语言中的结构体与共用体，分别举例说明它们的应用场合。

第 2 章　ARM Cortex-M0＋处理器

本章导读：KL 系列 MCU 的内核使用 ARM Cortex-M0＋处理器，需要学习 ARM Cortex-M0＋汇编的读者可以阅读本章全部内容，一般读者简要了解 2.1 节即可。虽然本书使用 C 语言阐述 ARM Cortex-M0＋ KL 系列 MCU 的嵌入式开发，但理解一两个结构完整、组织清晰的汇编程序对嵌入式开发将有很大帮助。实际上，通过一个单元时间学习本章内容，并没有耽误多少时间，但对理解机器码、理解一些细节非常有益！可以减少"书到用时方恨少"的场景。第 4 章中将结合 GPIO 的应用给出汇编实例，供读者学习参考。实际上，一些如初始化、操作系统调度、快速响应等特殊功能必须使用汇编完成。本章给出 ARM Cortex-M0＋的特点、内核结构、存储器映像及内部寄存器概述；给出指令简表、寻址方式及指令的分类介绍；给出指令集与机器码对应表，供机器码级别的调试分析使用；给出 ARM Cortex-M0＋汇编语言的基本语法。

本章参考资料：2.1.1 节的 ARM Cortex-M0＋处理器特点与结构图及 2.1.3 节的 M0＋的寄存器参考自《M0＋用户指南》；2.4 节的 ARM Cortex-M0＋汇编语言的基本语法，参考自《GNU 汇编语法》《Kinetis 汇编参考手册》及《M0＋用户指南》。

2.1　ARM Cortex-M0＋处理器简介

在 1.1.2 节中谈及嵌入式系统发展简史时，已经简要介绍了 ARM。本书以 KL 系列 MCU 阐述嵌入式应用，该系列的内核①使用 32 位 ARM Cortex-M0＋处理器（简称 CM0＋），它是 ARM 大家族中的重要一员，目标是取代传统 8/16 位处理器市场。本章简要阐述 CM0＋处理器。

CM0＋系列处理器是 2012 年推出的，主要目标市场是 8 位/16 位微控制器的升级换代，具有性价比高、功耗低等特点。了解其特点、内核结构、存储器映像、内部寄存器、寻址方式及指令系统，可以为进一步学习和应用 CM0＋打下基础。

2012 年 3 月 14 日，ARM 公司于中国上海发布了一款拥有全球最低功耗的微处理器 CM0＋。该处理器采用低成本的 90nm 低功耗（Low Power，LP）工艺，耗电量仅为 9μA/MHz；该微处理器基于 ARMv6M 架构支持 Thumb 指令集②和部分 Thumb2 指令集③；加入多个重要新特性，包括单周期输入输出（IO）以加快通用输入输出（GPIO）和外围设备的存取速度、改良的调试和追踪能力、二阶流水线技术以减少每个指令所需的时钟周期数

---

①　这里使用内核（Core）一词，而不用 CPU，原因在于 ARM 中使用内核术语涵盖了 CPU 功能，它比 CPU 功能可扩充一些。一般情况下，可以认为两个术语概念等同。

②　Thumb 是 ARM 体系结构中一种 16 位的指令集。

③　Thumb2 是 ARM 体系结构中混合 32/16 位的指令集。

(CPI)和优化的闪存访问方式等，以进一步降低功耗。

Cortex-M0＋处理器不仅延续了易用性、C 语言编程模型等优势，而且能够兼容已有的 Cortex-M0 处理器的工具。作为 Cortex-M 处理器系列中的一员，Cortex-M0＋处理器同样可获得 ARM Cortex-M 整个系统的全面支持，其良好的软件兼容性，使其能够方便地被移植到更高性能的 Cortex-M3 或 Cortex-M4 等系列处理器。

2.1.1　ARM Cortex-M0＋处理器内部结构概要

Cortex-M0＋处理器组件结构图见图 2-1，图中虚线表示可选组件，下面简要介绍各部分。

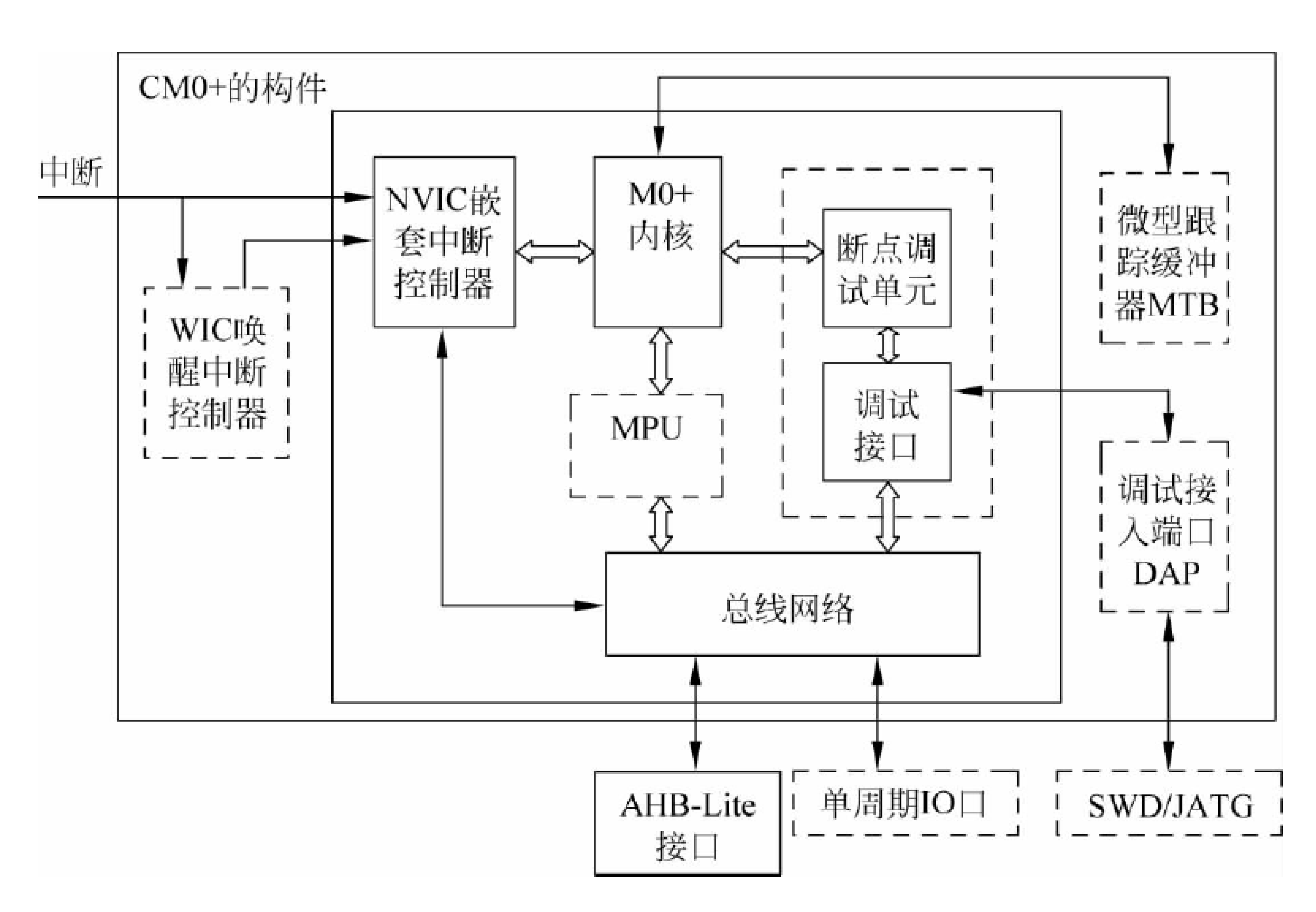


图 2-1　ARM Cortex-M0＋处理器结构图

1. M0＋内核

32 位 ARM Cortex-M0＋处理器是以一个“处理器子系统”的形式出现的，其 CPU 内核本身与 NVIC 和一系列调试块都亲密耦合。Cortex-M0＋处理器基于 2 级流水线冯·诺依曼架构[①]，内核版本为 ARMv6-M，支持 16 位 Thumb 指令集，同时采用 Thumb2 技术。M0＋内核的性能是接近 8 位或 16 位竞争产品 CoreMark/mA[②] 的二倍。

2. 嵌套中断向量控制器

NVIC(Nested Vectored Interrupt Controller)是一个在 CM0＋中内建的中断控制器。中断的具体路数由芯片厂商定义，本书使用的 MCU(恩智浦公司以 CM0＋为内核设计的微处理器系列之一)共有 32 个可屏蔽外部中断源并支持 4 级优先级。NVIC 是与 CPU 紧耦合的，它还包含若干个系统控制寄存器。因为 NVIC 支持中断嵌套，使得在 M0＋上处理嵌

① Cortex-M3/M4 采用哈佛结构，而 Cortex-M0＋采用的是冯·诺依曼结构。区别在于它们是不是具有独立的程序指令存储空间和数据存储地址空间，如果有的话，就是哈佛结构，如果没有就是冯·诺依曼结构。而具有独立的地址空间也就意味着在地址总线和控制总线上至少要有一种总线必须是独立的，这样才能保证地址空间的独立性。

② CoreMark 是一项基准测试，它的目标就是要测试处理器核心性能。CoreMark 能分析并为处理器管线架构和效率评分，CoreMark 已成为量测与比较处理器性能的业界标准基准测试。CoreMark 数字越高，意味着更高的性能。

套中断时清晰而强大。它还采用了向量中断的机制,在中断发生时,它会自动取出对应的服务例程入口地址,并且直接调用,无须软件判定中断源,缩短中断延时。为优化低功耗设计,NVIC嵌套中断控制器还集成一个可选WIC(唤醒中断控制器),在睡眠模式或深度睡眠模式下,芯片可快速进入超低功耗状态,且只能被WIC唤醒源唤醒。CM+的构件中,还包含一个24位倒计时定时器SysTick,即使系统在睡眠模式下也能工作,作为嵌套中断向量控制器NVIC的一部分实现,若用作实时操作系统RTOS的时钟,将给RTOS在同类内核芯片间移植带来便利。

3. 总线网络

BusMatrix是M0+内部总线系统的核心。它是一个AHB[①]互连的网络,通过它可以让数据在不同的总线之间并行传送(只要两个总线主机不试图访问同一块内存区域)。BusMatrix还提供了附加的数据传送管理设施,包括一个写缓冲以及一个按位操作的逻辑,这个按位操作的逻辑被称为位带。

4. 调试组件

M0+处理器实现了一个完全基于硬件的调试解决方案。该调试方案通过2针脚串行线协议(SWD)访问处理器、内存和外设。能够支持两个硬件断点和两个观察点;支持单步调试和向量捕捉;支持多个软件断点;通过总线网络非侵入式访问内核外设和零等待系统从机,调试器甚至能够在处理器运行时,访问包括内存在内的设备;在处理器停止状态时,可完全访问内核寄存器组。

5. 总线接口

CM0+处理器提供一个基于被称之为高级微控制器总线体系结构的总线规范AMBA技术的单一32位系统接口,可以高速整体访问所有系统外设和内存。所谓总线规范AMBA(Advanced Microcontroller Bus Architecture)是一组针对基于ARM内核、片上系统之间通信而设计的标准协议。总线规范AMBA具有可选的32位单周期I/O接口,支持高速访问紧密耦合外设,如GPIO外设模块。该接口可从处理器、调试器以载入、存储方式访问,但不能执行代码;可选的32位从机接口,支持调试访问端口(Debug Access Port,DAP)。该端口可通过串行或JATG协议调试;可选的内存保护单元接口;可选的执行跟踪接口(Execution Trace Interface,ETI),可配置执行跟踪缓冲区执行跟踪组件功能。在AMBA总线规范中,定义了AHB、APB、ASB三种总线。

6. 其他模块

系统控制块(System Control Block,SCB)提供了系统运行信息和系统配置功能,包括配置、控制、报告系统异常等。微型跟踪缓冲器(MTB)提供程序追踪功能,可以产生指令用来访问变化的数据。存储器保护单元(MPU)是一个选配的单元,本书中介绍的芯片不包含此组件。

2.1.2　ARM Cortex-M0+处理器存储器映像

CM0+处理器直接寻址空间为4GB,地址范围是:0x0000_0000~0xFFFF_FFFF。这

① AHB(Advanced High Performace Bus,高性能总线)用于高性能系统模块的连接,支持突发模式数据传输和事务分割。

里所说的存储器映像，其含义是指，把这 4GB 空间当作存储器来看待，分成若干区间，都可安排一些什么实际的物理资源。在 3.3.1 节结合具体芯片，还将给出较详细的阐述。尽管如此，ARM 定出的条条框框是粗线条的，它依然允许芯片制造商灵活地分配存储器空间，以制造出各具特色的 MCU 产品。

图 2-2 给出了 M0＋的存储器空间地址映像。CM0＋只有一个单一固定的存储器映射。这一点极大地方便了软件在各种 Cortex-M0＋内核间的移植。举个简单的例子，各款 Cortex-M0＋核 MCU 的 NVIC 和 MPU 都在相同的位置布设寄存器，使得它们变得通用。

（1）它的存储器映射是预定义的，并且还规定好了哪个位置使用哪条总线。

（2）Cortex-M0＋的存储器系统支持所谓的"位带"操作。通过它，实现了对单一比特的原子操作①。位带操作仅适用于一些特殊的存储器区域中。

代码0.5GB	0x0000_0000 ~ 0x1FFF_FFFF
SRAM0.5GB	0x2000_0000 ~ 0x3FFF_FFFF
外设0.5GB	0x4000_0000 ~ 0x5FFF_FFFF
外部RAM 1GB	0x6000_0000 ~ 0x9FFF_FFFF
外设1GB	0xA000_0000 ~ 0xDFFF_FFFF
私有外设总线1MB	0xE000_0000 ~ 0xE00F_FFFF
系统保留511MB	0xE010_0000 ~ 0xFFFF_FFFF

图 2-2　M0＋的存储器空间地址映像

（3）Cortex-M0＋的存储器系统支持小端格式和大端格式②的配置。一般具体某款芯片在出厂时已经被厂商定义过。例如，本书中 KL25 芯片被配置为小端格式③。

2.1.3　ARM Cortex-M0＋处理器的寄存器

学习一个 CPU，理解其内部寄存器用途是重要一环。CM0＋处理器的寄存器有 R0～R15，如图 2-3 所示。其中，R13 作为堆栈指针 SP。SP 实质上有两个，但在同一时刻只能有一个可以看到，这也就是所谓的 banked 寄存器。特殊功能寄存器有预定义的功能，而且必须通过专用的指令来访问。

1. 通用寄存器 R0～R12

R0～R12 是最具"通用目的"的 32 位通用寄存器，用于数据操作，复位后初始值为随机值。32 位的 Thumb2 指令可以访问所有通用寄存器。但绝大多数 16 位 Thumb 指令只能访问 R0～R7。因而 R0～R7 又被称为低组寄存器，所有指令都能访问它们。R8～R12 也被称为高组寄存器。只有很少的 16 位 Thumb 指令能访问它们，32 位的指令则不受限制。

2. 堆栈指针 R13

R13 是堆栈指针。在 CM0＋处理器内核中共有两个堆栈指针，主堆栈指针 MSP 和进程堆栈指针 PSP，若用户用到其中一个，另一个必须用特殊指令来访问（MRS、MSR 指令），

---

① 详见《Cortex-M3 权威指南》，P83。本书 13.4 节阐述位带技术的应用方法。

② 字节存储顺序（Endianess）分为小端格式（Little Endian）和大端格式（Big Endian），所谓小端格式是指字的低字节存储在低地址中，字的高字节存储在高地址中。大端格式则反之。

③ 参见《KL25 参考手册》3.3 节表 3-4。

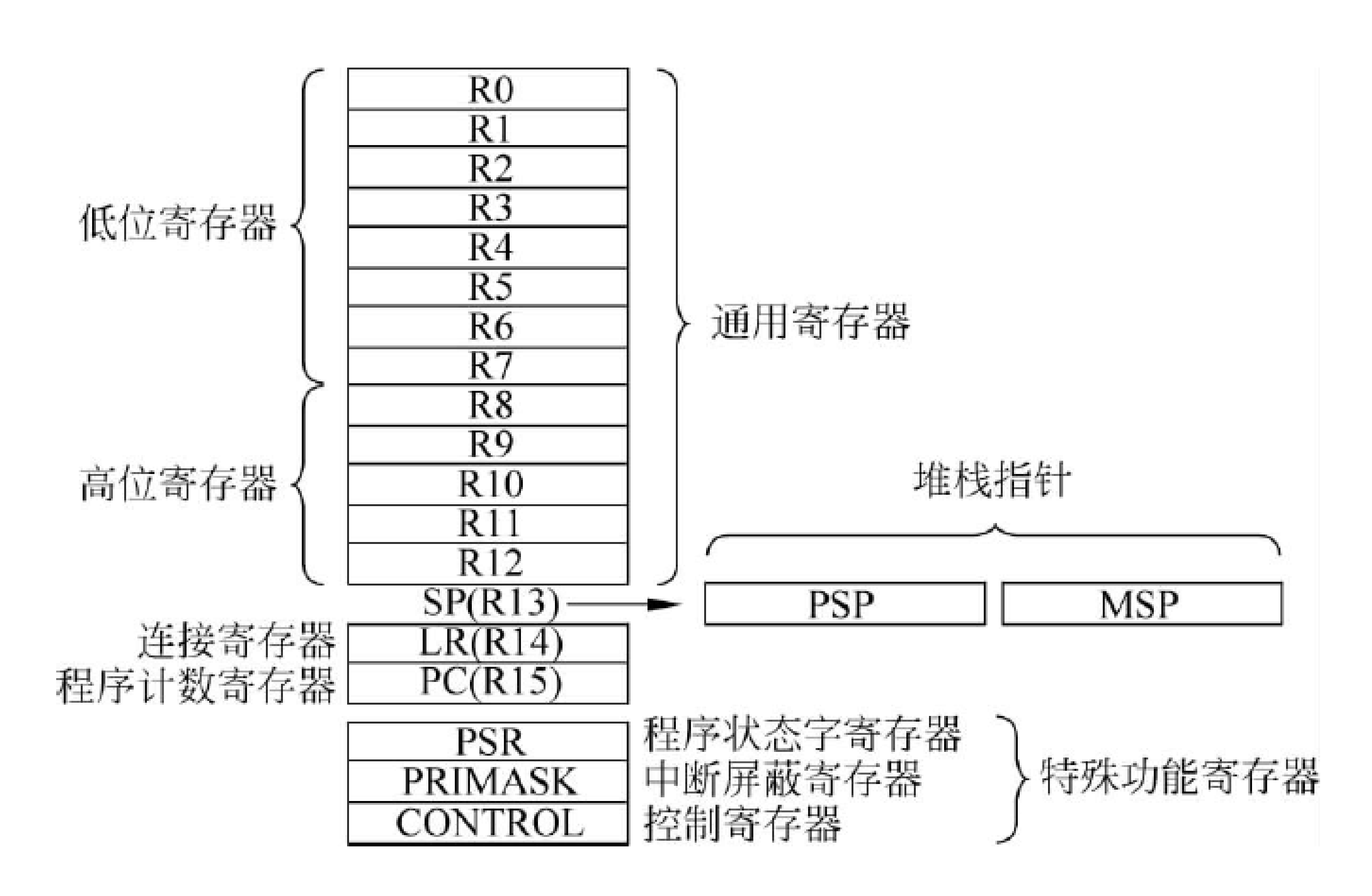


图 2-3 ARM Cortex-M0+处理器的寄存器组

因此任一时刻只能使用其中的一个。主堆栈指针 MSP 是复位后默认使用的堆栈指针,它可由操作系统内核、中断服务例程以及所有需要特权访问的应用程序代码来使用。进程堆栈指针 PSP 用于常规的应用程序代码(不处于中断服务例程中时),该堆栈一般供用户的应用程序代码使用。要注意的是,并不是每个应用工程都用到这两个堆栈指针。简单的应用程序只用 MSP 就够了,并且 PUSH 指令和 POP 指令默认使用 MSP(有时 MSP 直接记为 SP)。另外,堆栈指针的最低两位永远是 0,即堆栈总是 4 字节对齐的。

3. 连接寄存器 R14

当调用一个子程序时,由 R14 存储返回地址。不像大多数其他处理器,ARM 为了减少访问内存的次数(访问内存的操作往往要三个以上指令周期,带 MMU 和 Cache 的就更加不确定了),把返回地址直接存储在寄存器中。这样足以使很多只有 1 级子程序调用的代码无须访问内存(堆栈内存),从而提高了子程序调用的效率。如果多于 1 级,则需要把前一级的 R14 值压到堆栈里。在 ARM 上编程时,应尽量使用寄存器保存中间结果。

4. 程序计数寄存器 R15

R15 是程序计数器(Program Counter,PC),指向当前的程序地址。如果修改它的值,就能改变程序的执行流程(很多高级技巧隐藏其中)。在汇编代码中也可以使用名字"PC"来访问它。因为 CM0+内部使用了指令流水线,读 PC 时返回的值是当前指令的地址+4。CM0+中的指令至少是半字对齐的,所以 PC 的第 0 位总是 0。然而,在分支时,无论是直接写 PC 的值,还是使用分支指令,都必须保证加载到 PC 的数值是奇数(即第 0 位为 1),用以表明这是在 Thumb 状态下执行,倘若第 0 位为 0,则视为企图转入 ARM 模式,CM0+将产生异常。

5. 特殊功能寄存器

M0+内核包括一组特殊功能寄存器,包括程序状态字寄存器组(xPSR)、中断屏蔽寄存器(PRIMASK)和控制寄存器(CONTROL)。

1) 程序状态字寄存器

程序状态字寄存器在内部分为以下几个子寄存器:APSR、IPSR、EPSR。用户可以使用 MRS 和 MSR 指令访问寄存器。三个子寄存器既可以单独访问,也可以两个或三个组合到一起访问。使用三合一方式访问时,把该寄存器称为 xPSR,见表 2-1。

表 2-1　CM0＋程序状态寄存器(xPSR)

数据位	31	30	29	28	27～25	24	23～10	9	8～6	5	4	3	2	1	0
APSR	N	Z	C	V											
IPSR										异常号					
EPSR						T									
xPSR	N	Z	C	V		T				异常号					

(1) 应用程序状态寄存器(Application Program Status Register，APSR)：显示算术运算单元 ALU 状态位的一些信息。**负标志 N**：若结果最高位为 1，相当于有符号运算中结果为负，则置 1，否则清 0。**零标志 Z**：若结果为 0，则置 1，否则清 0。**进位标志 C**：若有最高位的进位(减法为借位)，则置 1，否则清 0。**溢出标志 V**：若溢出，则置 1，否则清 0。程序运行过程中这些位会根据运算结果而改变。在条件转移指令中也可被用到。复位之后，这些位是随机的。

(2) 中断程序状态寄存器(Interrupt Program Status Register，IPSR)：每次异常完成之后，处理器会实时更新 IPSR 内的异常号。只能被 MRS 指令读写。进程模式下，值为 0；Handler 模式[1]下，存放当前异常的异常号。复位之后，寄存器被自动清零。复位异常号是一个暂时值，复位时，是不可见的。

(3) 执行程序状态寄存器(Execution Program Status Register，EPSR)：T 标志位指示当前运行的是否是 Thumb 指令，该位是不能被软件读取的。运行复位向量对应的代码时置 1。如果该位为 0，会发生硬件异常，进入硬件中断服务例程。

2) 中断屏蔽寄存器

中断屏蔽寄存器 PRIMASK 的 D31～D1 位保留，只有 D0 位(记为 PM)有意义。当该位被置位时，除不可屏蔽中断和硬件错误之外的所有中断都被屏蔽。使用特殊指令(如 MSR，MRS)可以访问该寄存器，还有一条称为改变处理器状态的特殊指令 CPS 也能访问它。只有在实时任务时才会用到。执行汇编指令“CPSID i”，将 D0 位置 1(关总中断)；执行汇编指令“CPSIE i”，将 D0 位清 0(开总中断)，其中，i 代表 IRQ 中断。IRQ 是 Interrupt Request(非内核中断请求)的缩写。

3) 控制寄存器

内核中的控制寄存器 CONTROL 的 D31～D2 位保留，D1、D0 位含义如下。

D1(SPSEL)——堆栈指针选择位。SPSEL＝0，使用主堆栈指针 MSP 为当前堆栈指针(复位后默认值)；SPSEL＝1，在线程模式下，使用线程堆栈 PSP 指针为当前堆栈指针。特权、线程模式下，软件可以更新 SPSEL 位。在 Handler 模式下，写该位无效。复位后，控制寄存器清零。可用 MRS 指令读该寄存器，MSR 指令写该寄存器。非特权访问无效。

D0(nPRIV)——如果权限扩展，在线程模式下定义执行特权：nPRIV＝ 0，线程模式下可以特权访问；nPRIV＝1，线程模式下无特权访问。在 Handler 模式下，总是特权访问。

---

① 这里的 Handler 模式是指异常(中断)的模式。进程模式则指通常的程序执行过程，在一些操作系统下，也称任务模式。

2.2 ARM Cortex-M0+处理器的指令系统

CPU 的功能是从外部设备获得数据，通过加工、处理，再把处理结果送到 CPU 的外部世界。设计一个 CPU，首先需要设计一套可以执行特定功能的操作命令，这种操作命令称为**指令**。CPU 所能执行的各种指令的集合，称为该 CPU 的**指令系统**。表 2-2 给出了 ARM Cortex-M 指令集概况。

表 2-2　ARM Cortex-M 指令集概况

处理器型号	Thumb	Thumb2	硬件乘法	硬件除法	饱和运算	DSP 扩展	浮点	ARM 架构[①]	核心架构
Cortex-M0	大部分	子集	1 或 32 个周期	无	无	无	无	ARMv6M	冯·诺依曼
Cortex-M0+	大部分	子集	1 或 32 个周期	无	无	无	无	ARMv6M	冯·诺依曼
Cortex-M1	大部分	子集	3 或 33 个周期	无	无	无	无	ARMv6M	冯·诺依曼
Cortex-M3	全部	全部	1 个周期	有	有	无	无	ARMv7M	哈佛
Cortex-M4	全部	全部	1 个周期	有	有	有	可选	ARMv7E-M	哈佛

Cortex-M0+处理器将上一代 Cortex-M0 升级为真正的 8 位替代产品，同时保留了与所有其他 Cortex-M 级处理器之间的兼容性。其指令系统是 Cortex-M3/M4 指令集的一部分，并且与 Cortex-M0 完全兼容，这允许设计人员可重复利用其现有的编译器和调试工具。

2.2.1 ARM Cortex-M0+指令简表与寻址方式

1. ARM Cortex-M0+指令简表

学习和记忆 CM0+基本指令对理解处理器特性十分有益。这里给出的 CM0+指令简表可以方便读者记忆基本指令。

CM0+共有 57 条基本指令，依据不同的寻址方式形成 68 条具体指令。为了方便学习，将这些指令分为数据传送类、数据操作类(算术运算、逻辑运算、位操作、反转字节、扩展字节和移位)、跳转类和其他指令等 4 大类指令。本节分别介绍这些指令，并对每条具体指令进行统一编号。表 2-3 给出保留字的简明含义，供读者了解和记忆 CM0+指令。**这里大部分为英文简写，请学习时自行给出英文全称，方便理解与记忆。**

2. CM0+的寻址方式

指令是对数据的操作，通常把指令中所要操作的数据称为操作数，CM0+处理器所需的操作数可能来自寄存器、指令代码、存储单元。而确定指令中所需操作数的各种方法称为寻址方式(Addressing Mode)。下面的指令格式中的“{}”表示其中可选项，如 LDRH Rt，[Rn {，#imm}]，表示有：“LDRH Rt，[Rn]”“LDRH Rt，[Rn，#imm]”两种指令格式。指令中的“[]”表示其中内容为地址。“@”表示注释。

---

① 架构(Architecture)，即体系结构，这里主要指使用的指令集。由同一架构，可以衍生出许多不同处理器型号。对 ARM 而言，其他芯片厂商，可由 ARM 提供的一种处理器型号具体生产出许多不同的 MCU 或应用处理器型号。ARMv6-M 是一种架构型号，其中 v6 是指版本号，而基于该架构处理器有 Cortex-M0、Cortex-M0+、Cortex-M1 等。一般读者了解基本脉络即可。

表 2-3　CM0＋指令简表

类　型		保　留　字	含　义
数据传送类		ADR	生成与 PC 指针相关的地址
		LDR、LDRH、LDRB、LDRSB、LDRSH、LDM	存储器中内容加载到寄存器中
		STR、STRH、STRB、STM	寄存器中内容存储至存储器中
		MOV、MVN	寄存器间数据传送指令
		PUSH、POP	进栈、出栈
数据操作类	算术运算类	ADC、ADD、RSB、SBC、SUB、MUL	加、减、乘指令
		CMN、CMP	比较指令
	逻辑运算类	AND、ORR、EOR、BIC	按位与、或、异或、位段清零
	数据序转类	REV、REV16、REVSH	反转字节序
	扩展类	SXTB、SXTH、UXTB、UXTH	无符号扩展字节、有符号扩展字节
	位操作类	TST	测试位指令
	移位类	ASR、LSL、LSR、ROR	算术右移；逻辑左移、逻辑右移、循环右移
跳转控制类		B{cc}、BL、BX、BLX	跳转指令
其他指令		BKPT、CPSID、CPSIE、DSB、DMB、ISB、MRS、MSR、NOP、SEV、SVC、WFE、WFI	

1）立即数寻址

在立即数寻址方式中，操作数直接通过指令给出，数据包含指令编码中，随着指令一起被编译成机器码存储于程序空间中。用“#”作为立即数的前导标识符。CM0＋的立即数范围是 0x00～0xff。例如：

```
SUB   R1,R0,#1       @R1←R0−1
MOV   R0,#0xff       @将立即数 0xff 装入 R0 寄存器
```

2）寄存器寻址

在寄存器寻址中，操作数来自于寄存器。例如：

```
MOV   R1,R2         @R1 ←R2
SUB   R0,R1,R2      @R0←R1−R2
```

在存数与取数指令中，LDM、STM 可以一次操作多个寄存器。有的资料称为“多寄存器寻址”，当然，也属于寄存器寻址，不必另归一类，只是寄存器寻址的一种表现形式。

3）偏移寻址及寄存器间接寻址

在偏移寻址中，操作数来自于存储单元，指令中通过寄存器及偏移量给出存储单元的地址。偏移量不超过 4KB(指令编码中偏移量为 12 位)。偏移量为 0 的偏移寻址也称为寄存器间接寻址。例如：

```
LDR   R3, [PC, #2]      @地址为(PC + 2)的存储器单元内容加载到 R3 中
LDR   R3,[R4]           @地址为 R4 的存储单元内容加载到 R3 中
```

4）直接寻址

在直接寻址方式中，操作数来自于存储单元，指令中直接给出存储单元地址。指令码

中,显式给出数据的位数,有字(4字节)、半字(2字节)、单字节三种情况。例如:

```
LDR   Rt,label     @从标号 label 处连续取 4 字节至 Rt 寄存器中
LDRH  Rt,label     @从地址 label 处读取半字到 Rt
LDRB  Rt,label     @从地址 label 处读取字节到 Rt
```

2.2.2 数据传送类指令

数据传送类指令的功能有两种情况,一是取存储器地址空间中的数传送到寄存器中,二是将寄存器中的数传送到另一寄存器或存储器地址空间中。M0+数据传送类基本指令有16条。

1. 取数指令

存储器中内容加载到寄存器中的指令见表2-4,其中,LDR、LDRH、LDRB指令分别表示加载来自存储器单元的一个字、半字和单字节(不足部分以0填充)。LDRSH和LDRSB指令指加载存储单元的半字、字节有符号数扩展成32位到指定寄存器Rt。

表 2-4 取数指令

编号	指　令	说　明
(1)	LDR Rt,[<Rn|SP>{,#imm}]	从{SP/Rn+#imm}地址处,取字到Rt,imm=0,4,8,…,1020
	LDR Rt,[Rn,Rm]	从地址Rn+Rm处读取字到Rt
	LDR Rt,label	从label指定的存储器单元取数至寄存器,label必须在当前指令的−4～4KB范围内,且应4字节对齐
(2)	LDRH Rt,[Rn{,#imm}]	从{Rn+#imm}地址处,取半字到Rt中,imm=0,2,4,…,62
	LDRH Rt,[Rn,Rm]	从地址Rn+Rm处读取半字到Rt
(3)	LDRB Rt,[Rn{,#imm}]	从{Rn+#imm}地址处,取字节到Rt中,imm=0～31
	LDRB Rt,[Rn,Rm]	从地址Rn+Rm处读取字节到Rt
(4)	LDRSH Rt,[Rn,Rm]	从地址Rn+Rm处读取半字至Rt,并带符号扩展至32位
(5)	LDRSB Rt,[Rn,Rm]	从地址Rn+Rm处读取字节至Rt,并带符号扩展至32位
(6)	LDM Rn{!},reglist	从Rn处读取多个字,加载到reglist列表寄存器中,每读一个字后Rn自增一次

在LDM Rn{!},reglist指令中,Rn表示存储器单元起始地址的寄存器;reglist可包含一个或多个寄存器,若包含多个寄存器必须以“,”分隔,外面用“{}”标识;“!”是一个可选的回写后缀,reglist列表中包含Rn寄存器时不要回写后缀,否则须带回写后缀“!”。带后缀时,在数据传送完毕之后,将最后的地址写回到Rn=Rn+4×(n−1),n为reglist中寄存器的个数。Rn不能为R15,reglist可以为R0～R15任意组合;Rn寄存器中的值必须字对齐。这些指令不影响N、Z、C、V状态标志。

2. 存数指令

寄存器中内容存储至存储器中的指令见表2-5。STR、STRH和STRB指令存储Rt寄存器中的字、低半字或低字节至存储器单元。存储器单元地址由Rn与Rm之和决定。Rt、Rn和Rm必须为R0～R7之一。

其中,STM Rn!, reglist指令将reglist列表寄存器内容以字存储至Rn寄存器中的存

储单元地址。以 4 字节访问存储器地址单元，访问地址从 Rn 寄存器指定的地址值到 Rn＋4×(n−1)，n 为 reglist 中寄存器的个数。按寄存器编号递增顺序访问，最低编号使用最低地址空间，最高编号使用最高地址空间。对于 STM 指令，若 reglist 列表中包含 Rn 寄存器，则 Rn 寄存器必须位于列表首位。如果列表中不包含 Rn，则将位于 Rn＋4×n 的地址回写到 Rn 寄存器中。这些指令不影响 N、Z、C、V 状态标志。

表 2-5　存数指令

编号	指　　令	说　　明
(7)	STR Rt, [<Rn | SP>{, #imm}]	把 Rt 中的字存储到地址 SP/Rn＋#imm，imm＝0,4,8,…,1020
	STR Rt, [Rn, Rm]	把 Rt 中的字存储到地址 Rn＋Rm 处
(8)	STRH Rt, [Rn {, #imm}]	把 Rt 中的低半字存储到地址 SP/Rn＋#imm，imm＝0,2,4,…,62
	STRH Rt, [Rn, Rm]	把 Rt 中的低半字存储到地址 Rn＋Rm 处
(9)	STRB Rt, [Rn {, #imm}]	把 Rt 中的低字节存储到 SP/Rn＋#imm。imm＝0～31
	STRB Rt, [Rn, Rm]	把 Rt 中的低字节存储到地址 Rn＋Rm 处
(10)	STM Rn!, reglist	存储多个字到 Rn 处。每存一个字后 Rn 自增一次

3. 寄存器间数据传送指令

MOV 指令(见表 2-6)，Rd 表示目标寄存器；imm 为立即数，范围为 0x00～0xff。当 MOV 指令中 Rd 为 PC 寄存器时，丢弃第 0 位；当出现跳转时，传送值的第 0 位清零后的值作为跳转地址。虽然 MOV 指令可以用作分支跳转指令，但强烈推荐使用 BX 或 BLX 指令。这些指令影响 N、Z 状态标志，但不影响 C、V 状态标志。

表 2-6　寄存器间数据传送指令

编号	指　　令	说　　明
(11)	MOV Rd, Rm	Rd←Rm, Rd 只可以是 R0～R7
(12)	MOVS Rd, #imm	MOVS 指令功能与 MOV 相同，且影响 N、Z 标志
(13)	MVN Rd, Rm	将寄存器 Rm 中数据取反，传送给寄存器 Rd，影响 N、Z 标志

4. 堆栈操作指令

堆栈操作指令见表 2-7。PUSH 指令将寄存器值存于堆栈中，最低编号寄存器使用最低存储地址空间，最高编号寄存器使用最高存储地址空间；POP 指令将值从堆栈中弹回寄存器，最低编号寄存器使用最低存储地址空间，最高编号寄存器使用最高存储地址空间。执行 PUSH 指令后，更新 SP 寄存器值 SP＝SP−4；执行 POP 指令后更新 SP 寄存器值 SP＝SP＋4。若 POP 指令的 reglist 列表中包含 PC 寄存器，在 POP 指令执行完成时跳转到该指针 PC 所指地址处。该值最低位通常用于更新 xPSR 的 T 位，此位必须置 1 确保程序正常运行。

表 2-7　堆栈操作指令

编号	指　　令	说　　明
(14)	PUSH reglist	进栈指令。SP 递减 4
(15)	POP reglist	出栈指令。SP 递增 4

例如：

```
PUSH {R0,R4-R7}        @将 R0,R4～R7 寄存器值入栈
PUSH {R2,LR}           @将 R2,LR 寄存器值入栈
POP {R0,R6,PC}         @出栈值到 R0,R6,PC 中,同时跳转至 PC 所指向的地址
```

5. 生成与指针 PC 相关地址指令

ADR 指令(见表 2-8)将指针 PC 值加上一个偏移量得到的地址写进目标寄存器中。若利用 ADR 指令生成的目标地址用于跳转指令 BX、BLX,则必须确保该地址最后一位为 1。Rd 为目标寄存器,label 为与指针 PC 相关的表达式。在该指令下,Rd 必须为 R0～R7,数值必须字对齐且在当前 PC 值的 1020 字节以内。此指令不影响 N、Z、C、V 状态标志。这条指令主要提供编译阶段使用,一般可看成一条伪指令。

表 2-8　ADR 指令

编号	指　　令	说　　明
(16)	ADR Rd, label	生成与指针 PC 相关地址,将 label 的相对于当前指令的偏移地址值与 PC 相加或者相减(label 有前后,即负、正)写入 Rd 中

2.2.3　数据操作类指令

数据操作主要指算术运算、逻辑运算、移位等。

1. 算术运算类指令

算术类指令有加、减、乘、比较等,见表 2-9。

表 2-9　算术类指令

编号	指　　令	说　　明
(17)	ADC {Rd,} Rn, Rm	带进位加法。Rd←Rn+Rm+C,影响 N、Z、C 和 V 标志位
(18)	ADD {Rd} Rn, <Rm | #imm>	加法。Rd←Rn+Rm,影响 N、Z、C 和 V 标志位
(19)	RSB {Rd,} Rn, #0	Rd←0−Rn,影响 N、Z、C 和 V 标志位(KDS 环境不支持)
(20)	SBC {Rd,}Rn, Rm	带借位减法。Rd←Rn−Rm−C,影响 N、Z、C 和 V 标志位
(21)	SUB {Rd} Rn, <Rm | #imm>	常规减法。Rd←Rn−Rm/#imm,影响 N、Z、C 和 V 标志位
(22)	MUL Rd, Rn Rm	常规乘法,Rd←Rn * Rm,同时更新 N、Z 状态标志,不影响 C、V 状态标志。该指令所得结果与操作数是否为无符号、有符号数无关。Rd、Rn、Rm 寄存器必须为 R0～R7,且 Rd 与 Rm 须一致
(23)	CMN Rn, Rm	加比较指令。Rn+Rm,更新 N、Z、C 和 V 标志,但不保存所得结果。Rn、Rm 寄存器必须为 R0～R7
(24)	CMP Rn, #imm CMP Rn, Rm	(减)比较指令。Rn−Rm/#imm,更新 N、Z、C 和 V 标志,但不保存所得结果。Rn、Rm 寄存器为 R0～R7,立即数 imm 范围为 0～255

加、减指令对操作数的限制条件，见表 2-10。

表 2-10　ADC、ADD、RSB、SBC 和 SUB 操作数限制条件

指令	Rd	Rn	Rm	imm	限制条件
ADC	R0～R7	R0～R7	R0～R7	—	Rd 和 Rn 必须相同
ADD	R0～R15	R0～R15	R0～PC	—	Rd 和 Rn 必须相同； Rn 和 Rm 不能同时指定为 PC 寄存器
	R0～R7	SP 或 PC	—	0～1020	立即数必须为 4 的整数倍
	SP	SP	—	0～508	立即数必须为 4 的整数倍
ADD	R0～R7	R0～R7	—	0～7	—
	R0～R7	R0～R7	—	0～255	Rd 和 Rn 必须相同
	R0～R7	R0～R7	R0～R7	—	—
RSB	R0～R7	R0～R7	—	—	—
SBC	R0～R7	R0～R7	R0～R7	—	Rd 和 Rn 必须相同
SUB	SP	SP	—	0～508	立即数必须为 4 的整数倍
SUB	R0～R7	R0～R7	—	0～7	—
	R0～R7	R0～R7	—	0～255	Rd 和 Rn 必须相同
	R0～R7	R0～R7	R0～R7	—	—

2. 逻辑运算类指令

逻辑运算类指令见表 2-11。AND、EOR 和 ORR 指令把寄存器 Rn、Rm 值逐位与、异或和或操作；BIC 指令是将寄存器 Rn 的值与 Rm 的值的反码按位作逻辑“与”操作，结果保存到 Rd。这些指令更新 N、Z 状态标志，不影响 C、Z 状态标志。

表 2-11　逻辑运算类指令

编号	指令	说明	举例
(25)	AND {Rd,} Rn, Rm	按位与	AND R2, R2, R1
(26)	ORR {Rd,} Rn, Rm	按位或	ORR R2, R2, R5
(27)	EOR {Rd,} Rn, Rm	按位异或	EOR R7, R7, R6
(28)	BIC {Rd,} Rn, Rm	位段清零	BIC R0, R0, R1

Rd、Rn 和 Rm 必须为 R0～R7，其中，Rd 为目标寄存器，Rn 为存放第一个操作数寄存器且必须和目标寄存器 Rd 一致(即 Rd 就是 Rn)，Rm 为存放第二个操作数寄存器。

3. 移位类指令

移位类指令见表 2-12。ASR、LSL、LSR 和 ROR 指令，将寄存器 Rm 的值由寄存器 Rs 或立即数 imm 决定移动位数，执行算术右移、逻辑左移、逻辑右移和循环右移。这些指令中，Rd、Rm、Rs 必须为 R0～R7。对于非立即数指令，Rd 和 Rm 必须一致。Rd 为目标寄存器，若省去 Rd，表示其值与 Rm 寄存器一致；Rm 为存放被移位数据寄存器；Rs 为存放移位长度寄存器；imm 为移位长度，ASR 指令移位长度范围为 1～32，LSL 指令移位长度范围为 0～31，LSR 指令移位长度范围为 1～32。

表 2-12　移位指令

编号	指　　令	操　　作	举　　例
(29)	ASR {Rd,} Rm, Rs ASR {Rd,} Rm, #imm	→□□□□ …… □□□□→C b31　　b0	算术右移 ASR R7, R5, #9
(30)	LSL {Rd,} Rm, Rs LSL {Rd,} Rm, #imm	C←□□□□ …… □□□□←0 b31　　b0	逻辑左移 LSL R1, R2, #3
(31)	LSR {Rd,} Rm, Rs LSR {Rd,} Rm, #imm	0→□□□□ …… □□□□→C b31　　b0	逻辑右移 LSR R1, R2, #3
(32)	ROR {Rd,} Rm, Rs	→□□□□ …… □□□□→C b31　　b0	循环右移 ROR R4, R4, R6

1) 单向移位指令

算术右移指令 ASR 比较特别,它把要操作的字节当作有符号数,而符号位(b31)保持不变,其他位右移一位,即首先将 b0 位移入 C 中,其他位(b1～b31)右移一位,相当于操作数除以 2。为了保证符号不变,ASR 指令使符号位 b31 返回本身。逻辑右移指令 LSR 把 32 位操作数右移一位,首先将 b0 位移入 C 中,其他右移一位,0 移入 b31。根据结果,ASR、LSL、LSR 指令对标志位 N、Z 有影响;最后移出位更新 C 标志位。

2) 循环移位指令

在循环右移指令 ROR 中,将 b0 位移入 b31 中的同时也移入 C 中,其他位右移一位,从 b31～b0 内部看来循环右移了一位。根据结果,ROR 指令对标志位 N、Z 有影响;最后移出位更新 C 标志位。

4. 位测试指令

位操作类指令见表 2-13。

表 2-13　位测试指令

编号	指　　令	说　　明
(33)	TST Rn, Rm	将 Rn 寄存器值逐位与 Rm 寄存器值进行与操作,但不保存所得结果。为测试寄存器 Rn 某位为 0 或 1,将 Rn 寄存器某位置 1,其余位清零。寄存器 Rn、Rm 必须为 R0～R7。该指令根据结果,更新 N、Z 状态标志,但不影响 C、V 状态标志

5. 数据序转指令

数据序转指令见表 2-14。该指令用于改变数据的字节顺序。Rn 为源寄存器,Rd 为目标寄存器,且必须为 R0～R7 之一。REV 指令将 32 位大端数据转小端存放或将 32 位小端数据转大端存放;REV16 指令将一个 32 位数据划分成两个 16 位大端数据,将这两个 16 位大端数据转小端存放或将一个 32 位数据划分成两个 16 位小端数据,将这两个 16 位小端数据转大端存放;REVSH 指令将 16 位带符号大端数据转成 32 位带符号小端数据或将 16 位带符号小端数据转成 32 位带符号大端数据,如图 2-4 所示。这些指令不影响 N、Z、C、V 状态标志。

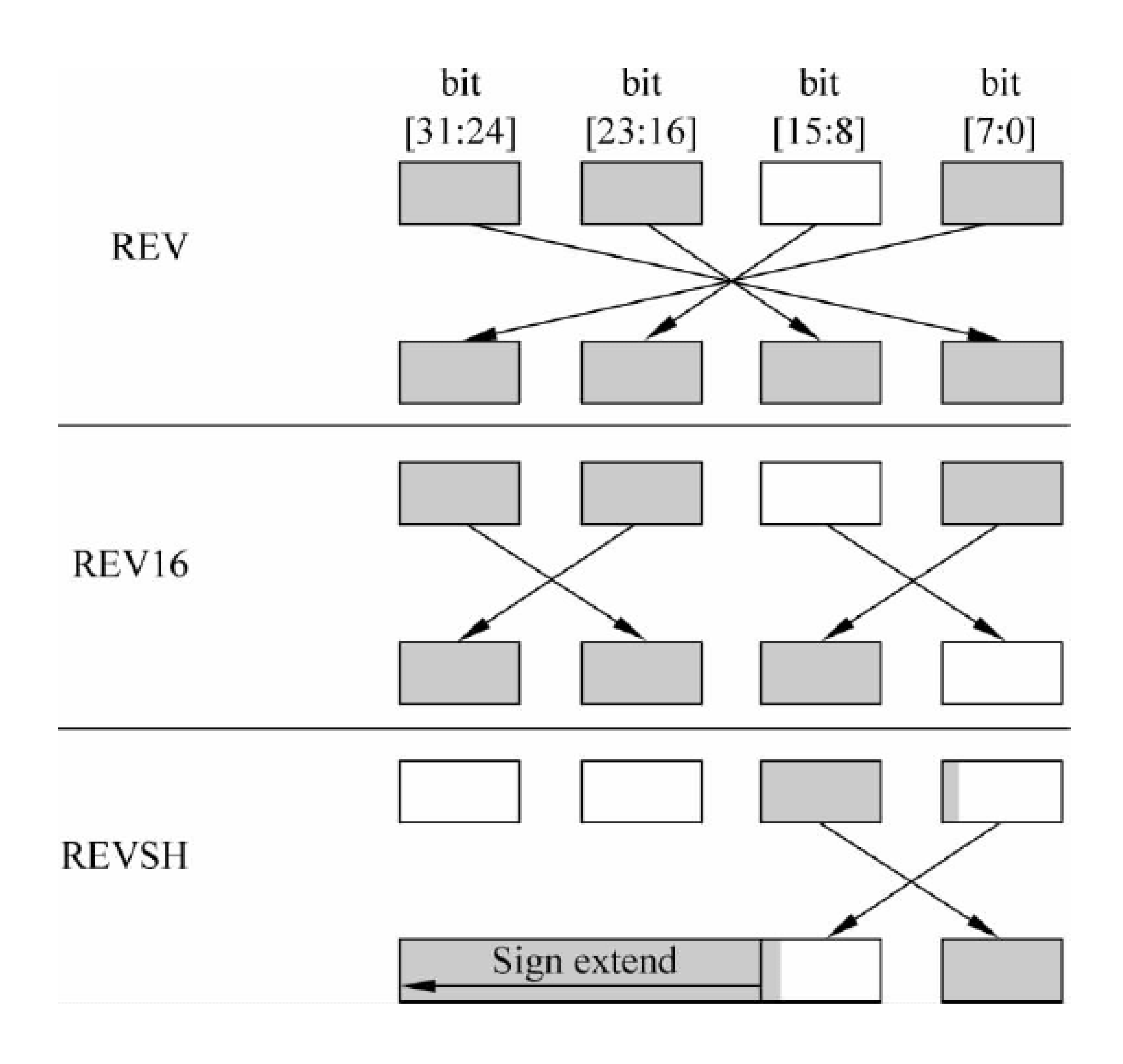


图 2-4　反序操作

表 2-14　数据序转指令

编号	指　令	说　明
(34)	REV Rd, Rn	将 32 位大端数据转小端存放或将 32 位小端数据转大端存放
(35)	REV16 Rd, Rn	将一个 32 位数据划分成两个 16 位大端数据，将这两个 16 位大端数据转小端存放或将一个 32 位数据划分成两个 16 位小端数据，将这两个 16 位小端数据转大端存放
(36)	REVSH Rd, Rn	将 16 位带符号大端数据转成 32 位带符号小端数据或将 16 位带符号小端数据转成 32 位带符号大端数据

6. 扩展类指令

扩展类指令见表 2-15。寄存器 Rm 存放待扩展操作数；寄存器 Rd 为目标寄存器；Rm、Rd 必须为 R0～R7。这些指令不影响 N、Z、C、V 状态标志。

表 2-15　扩展类指令

编号	指　令	说　明
(37)	SXTB Rd, Rm	将操作数 Rm 的 bit[7:0]带符号扩展到 32 位，结果保存到 Rd 中
(38)	SXTH Rd, Rm	将操作数 Rm 的 bit[15:0]带符号扩展到 32 位，结果保存到 Rd 中
(39)	UXTB Rd, Rm	将操作数 Rm 的 bit[7:0]无符号扩展到 32 位，结果保存到 Rd 中
(40)	UXTH Rd, Rm	将操作数 Rm 的 bit[15:0]无符号扩展到 32 位，结果保存到 Rd 中

2.2.4　跳转控制类指令

跳转控制类指令见表 2-16，这些指令不影响 N、Z、C、V 状态标志。

表 2-16　跳转控制类指令

编号	指　　令	跳 转 范 围	说　　明
(41)	B{cond} label	−256B～+254B	转移到 label 处对应的地址处。可以带(或不带)条件,所带条件见表 2-17,如:BEQ 表示标志位 Z=1 时转移
(42)	BL label	−16MB～+16MB	转移到 label 处对应的地址,并且把转移前的下条指令地址保存到 LR,并置寄存器 LR 的 bit[0]为 1,保证了随后执行 POP {PC}或 BX 指令时成功返回分支
(43)	BX Rm	任意	转移到由寄存器 Rm 给出的地址,寄存器 Rm 的 bit[0]必须为 1,否则会导致硬件故障
(44)	BLX Rm	任意	转移到由寄存器 Rm 给出的地址,并且把转移前的下条指令地址保存到 LR。寄存器 Rm 的 bit[0]必须为 1,否则会导致硬件故障

跳转控制类指令举例如下,特别注意 BL 用于调用子程序。

```
BEQ   label      @条件转移,标志位 Z=1 时转移到 label
BL    funC       @调用子程序 funC,把转移前的下条指令地址保存到 LR
BX    LR         @返回到函数调用处
```

B 指令所带条件众多,形成不同条件下的跳转,但只在前 256 至后 254 字节地址范围内跳转。B 指令所带的条件见表 2-17。

表 2-17　B 指令所带的条件

条件后缀	标志位	含　　义	条件后缀	标　志　位	含　　义
EQ	Z=1	相等	HI	C=1 并且 Z=0	无符号数大于
NE	Z=0	不相等	LS	C=1 或 Z=1	无符号数小于或等于
CS 或者 HS	C=1	无符号数大于或等于	GE	N=V	带符号数大于或等于
CC 或者 LO	C=0	无符号数小于	LT	N!=V	带符号数小于
MI	N=1	负数	GT	Z=0 并且 N=V	带符号数大于
PL	N=0	正数或零	LE	Z=1 并且 N!=V	带符号数小于或等于
VS	V=1	溢出	AL	任何情况	无条件执行
VC	V=0	未溢出			

2.2.5　其他指令

未列入数据传输类、数据操作类、跳转控制类三大类的指令,归为其他指令这一类。其他指令如表 2-18 所示。其中,spec_reg 表示特殊寄存器:APSR、IPSR、EPSR、IEPSR、IAPSR、EAPSR、PSR、MSP、PSP、PRIMASK 或者 CONTROL。

表中的中断指令(禁止总中断指令“CPSIE　i”,使能总中断指令“CPSID　i”)为编程必用指令,实际编程时,由宏函数给出。

下面对两条休眠指令 WFE 与 WFI 作简要说明。这两条指令均只用于低功耗模式,并不产生其他操作(这一点类似于 NOP 指令)。休眠指令 WFE 执行情况由事件寄存器决定。若事件寄存器为零,只有在发生如下事件时才执行:①发生异常,且该异常未被异常屏蔽寄

存器或当前优先级屏蔽；②在进入异常期间，系统控制寄存器的 SEVONPEND 置位；③若使能调试模式时，触发调试请求；④外围设备发出一个事件或在多重处理器系统中另一个处理器使用 SVC 指令。若事件寄存器为 1，WFE 指令清该寄存器后立刻执行。休眠指令 WFI 执行条件为：发生异常，或 PRIMASK. PM 被清 0，产生的中断将会先占，或发生触发调试请求(不论调试是否被使能)。

表 2-18　其他指令

类　　型	编号	指　　令	说　　明
断点指令	(45)	BKPT ＃imm	如果调试被使能，则进入调试状态(停机)。或者如果调试监视器异常被使能，则调用一个调试异常，否则调用一个错误异常。处理器忽视立即数 imm，立即数范围为 0～255，表示断点调试的信息。不影响 N、Z、C、V 状态标志
中断指令	(46)	CPSIE i	除了 NMI，使能总中断，不影响 N、Z、C、V 标志
	(47)	CPSID i	除了 NMI，禁止总中断，不影响 N、Z、C、V 标志
屏蔽指令	(48)	DMB	数据内存屏蔽(与流水线、MPU 和 cache 等有关)
	(49)	DSB	数据同步屏蔽(与流水线、MPU 和 cache 等有关)
	(50)	ISB	指令同步屏蔽(与流水线、MPU 等有关)
特殊寄存器操作指令	(51)	MRS Rd，spec_reg1	加载特殊功能寄存器值到通用寄存器。若当前执行模式不为特权模式，除 APSR 寄存器外，读其余所有寄存器值为 0
	(52)	MSR spe_reg，Rn	存储通用寄存器的值到特殊功能寄存器。Rd 不允许为 SP 或 PC 寄存器，若当前执行模式不为特权模式，除 APSR 外，任何试图修改寄存器的操作均被忽视。影响 N、Z、C、V 标志
空操作	(53)	NOP	空操作，但无法保证能够延迟时间，处理器可能在执行阶段之前就将此指令从线程中移除。不影响 N、Z、C、V 标志
发送事件指令	(54)	SEV	发送事件指令。在多处理器系统中，向所有处理器发送一个事件，也可置位本地寄存器。不影响 N、Z、C、V 标志
操作系统服务调用指令	(55)	SVC ＃imm	操作系统服务调用，带立即数调用代码。SVC 指令触发 SVC 异常。处理器忽视立即数 imm，若需要，该值可通过异常处理程序重新取回，以确定哪些服务正在请求。执行 SVC 指令期间，当前任务优先级高于等于 SVC 指令调用处理程序时，将产生一个错误。不影响 N、Z、C、V 标志
休眠指令	(56)	WFE	休眠并且在发生事件时被唤醒。不影响 N、Z、C、V 标志
	(57)	WFI	休眠并且在发生中断时被唤醒。不影响 N、Z、C、V 标志

2.3　ARM Cortex-M0＋指令集与机器码对应表

列出 CM0＋指令集与机器码对应表，目的是在一些深入细致的调试分析中，需要分析机器码，有了这张表便于进行这项工作。初学者了解即可。

CM0＋处理器指令集与机器码对应关系如表 2-19 所示。“机器码”列中，v 代表 immed_value；n 代表 Rn；m 代表 Rm；s 代表 Rs；r 代表 register_list；c 代表 condition；d 代表 Rd；l 代表 label。机器码均为大端对齐方式，即高位字节在低地址中，便于从左到右顺序阅

读。从表中看出,M0+大部分指令为16位,只有极少部分为32位。

表 2-19　指令集与机器码对应表

分类	序号	助记符	指令格式	机器码	实例	
					指令	机器码
数据传送类指令	1	LDR	LDR Rd,[RN,RM]	0101 100m mmnn nddd	LDR r4, [r4, r5]	5964
			LDR Rd,label	0100 1ddd vvvv vvvv	LDR r5, =runpin	4D0A
			LDR Rd,[< RN|SP >{,#imm}]	0110 1vvv vvnn nddd	LDR r1, [r5, #0]	6829
	2	LDRH	LDRH Rd,[Rn{,#imm}]	1000 1vvv vvnn nddd	LDRH r4, [r4, #30]	8be4
			LDRH Rd,[Rn,Rm]	0101 101m mmnn nddd	LDRH r4, [r4, r5]	5b64
	3	LDRB	LDRB Rd,[Rn{,#imm}]	0111 1vvv vvnn nmmm	LDRB r4, [r4, #30]	7FA4
			LDRB Rd,[Rn,Rm]	0101 110m mmnn nddd	LDRB r4, [r4, r5]	5D64
	4	LDRSH	LDRSH Rd,[Rn,Rm]V	0101 111m mmnn nddd	LDRSH r4, [r4, r5]	5F64
	5	LDRSB	LDRSB Rd,[Rn,Rm]	0101 011m mmnn nddd	LDRSB r4, [r4, r5]	5764
	6	LDM	LDM Rn{!},reglist	1100 1nnn rrrr rrrr	LDM r0,{r0,r3,r4}	C819
	7	STR	STR Rd, [< RN|SP >{,#imm}]	0110 0vvv vvnn nddd	STR r0, [r5, #4]	6068
			STR Rd,[Rn,Rm]	0101 000m mmnn nddd	STR r0,[r5,r4]	5128
	8	STRH	STRH Rd,[Rn{,#imm}]	1000 0vvv vvnn nddd	STRh r0,[r5,#4]	80A8
			STRH Rd,[Rn,Rm]	0101 001m mmnn nddd	STRh r0,[r5,r4]	5328
	9	STRB	STRB Rd,[Rn{,#imm}]	0111 0vvv vvnn nddd	STRB r0,[r5,#4]	7128
			STRB Rd,[Rn,Rm]	0101 010m mmnn nddd	STRB r0,[r5,r4]	5528
	10	STM	STM Rn!,reglist	1100 0nnn rrrr rrrr	STMIA r0,{r0,r3,r4}	C019
	11	MOV	MOV Rd,Rm	0001 1100 00nn nddd	MOV r1,r2	1C11
	12	MOVS	MOVS Rd,#imm	0010 0ddd vvvv vvvv	MOVS r1,#8	2108
	13	MVN	MVN Rd,Rm	0100 0011 11mm mddd	MVN r1,r3	43D9
	14	PUSH	PUSH reglist	1011 010R rrrr rrrr	PUSH {r1}	B402
	15	POP	POP reglist	1011 110R rrrr rrrr	PUSH {r1}	BC02
	16	ADR	ADR Rd,label	1010 0ddd vvvv vvvv	ADR R3,REPEAT	A305
数据操作类指令	17	ADC	ADC {Rd} Rn,Rm	0100 0001 01mm mddd	ADC r2,r3	415A
	18	ADD	ADD {Rd} Rn < Rm|#imm >	0001 100m mmnn nddd	ADD r2,r3	18D2
			ADD Rn,#imm	0011 0ddd vvvv vvvv	ADD r2,#12	320C
			ADD Rd,Rn,#imm	0001 110v vvnn nddd	ADD r2,r3,#1	1C5A
			ADD Rd,Rn,#imm	0001 100m mmnn nddd	ADD r2,r3,r4	191A
	19	RSB	(KDS环境不支持)			
	20	SBC	SBC {Rd,} Rn, Rm	0100 0001 10mm mnnn	SBC R7,R7,R1	418F
	21	SUB	SUB Rd,#imm	0011 1ddd vvvv vvvv	SUB r4,#1	3C01
			SUB Rd,Rn,#imm	0001 111v vvnn nddd	SUB r3,r4,#1	1E63
			SUB Rd,Rn,Rm	0001 101m mmnn nddd	SUB r3,r4,r1	1A63
	22	MUL	MUL Rd,Rn,Rm	0100 0011 01mm mddd	MUL r1,r2,r1	4351
	23	CMN	CMN Rn,Rm	0100 0010 11mm mnnn	CMN r1,r2	42D1
	24	CMP	CMP Rn,#imm	0010 1nnn vvvv vvvv	CMP r4,#1	2C01
			CMP Rn,Rm	0100 0010 10mm mnnn	CMP r4,r1	428C
	25	AND	AND {Rd,}Rn,Rm	0100 0000 00mm mddd	AND r1,r2	4011
	26	ORR	ORR {Rd,}Rn,Rm	0100 0011 00mm mddd	ORR r1,r2	4311
	27	EOR	EOR {Rd,}Rn,Rm	0100 0000 01mm mddd	EOR r1,r2	4051
	28	BIC	BIC {Rd,}Rn,Rm	0100 0011 10mm mddd	BIC r1,r2	4391

续表

分类	序号	助记符	指令格式	机器码	实例	
					指令	机器码
数据操作类指令	29	REV	REV Rd,Rn	1011 1010 00nn nddd	REV r1,r2	BA11
	30	REV16	REV16 Rd,Rn	1011 1010 01nn nddd	REV16 r3,r3	BA5B
	31	REVSH	REVSH Rd,Rn	1011 1010 11nn nddd	REVSH r4,r3	BADC
	32	SXTB	SXTB Rd,Rm	1011 0010 01mm mddd	SXTB r4,r3	B25C
	33	SXTH	SXTH Rd,Rm	1011 0010 00mm mddd	SXTH r4,r3	B21C
	34	UXTB	UXTB Rd,Rm	1011 0010 11mm mddd	UXTB r4,r3	B2DC
	35	UXTH	UXTH Rd,Rm	1011 0010 10mm mddd	UXTH r4,r3	B29C
	36	TST	TST Rn,Rm	0100 0010 00mm mnnn	TST r1,r2	4211
	37	ASR	ASR {Rd,}Rm,Rs	0100 0001 00ss smmm	ASR r3,r3,r5	412B
			ASR {Rd,}Rm,#imm	0001 0vvv vvmm mddd	ASR r7,r5,#6	11AF
	38	LSL	LSL {Rd,}Rm,Rs	0100 0000 10ss smmm	LSL r5,r6	40B5
			LSL {Rd,}Rm, #imm	0000 0vvv vvmm mddd	LSL r5,r6,#6	01B5
	39	LSR	LSR {Rd,}Rm,Rs	0100 0000 11ss sddd	LSR r5,r6	40F5
			LSR {Rd,}Rm, #imm	0000 1vvv vvmm mddd	LSR r5,r6,#6	09B5
	40	ROR	ROR {Rd,}Rm,Rs	0100 0001 11ss sddd	ROR r5,r6	41F5
跳转类指令	41	B	B label	1110 0vvv vvvv vvvv	B repeat	E7F8
			B{cond} label	1110 cccc vvvv vvvv	BNE repeat	D1F8
	42	BL	BL label	32 位指令	BL loop	F7FF FFD2
	43	BX	BX Rm	0100 0111 00mm m000	BX r1	4708
	44	BLX	BLX Rm	0100 0111 10mm m000	BLX r1	4788
其他指令	45	BKPT	BKPT #imm	1011 1110 vvvv vvvv	BKPT	BE00
	46	CPSIE	CPSIE i	1011 0110 0110 0010	CPSIE i	B662
	47	CPSID	CPSID i	1011 0110 0111 0010	CPSID i	B672
	48	DMB	DMB	32 位指令	DMB	F3BF 8F5F
	49	DSB	DSB	32 位指令	DSB	F3BF 8F4F
	50	ISB	ISB	32 位指令	ISB	F3BF 8F6F
	51	MRS	MRS Rd,spec_reg1	32 位指令	MRS R5, PRIMASK	F3EF 8510
	52	MSR	MSR spec_reg,Rn	32 位指令	MSR PRIMASK,R5	F385 8810
	53	NOP	NOP		NOP	46C0
	54	SEV	SEV	1011 1111 0100 0000	SEV	BF40
	55	SVC	SVC #imm	1101 1111 vvvv vvvv	SVC #12	DF0C
	56	WFE	WFE	1011 1111 0010 0000	WFE	BF20
	57	WFI	WFI	1011 1111 0011 0000	WFI	BF30

2.4 GNU 汇编语言的基本语法

能够在 MCU 内直接执行的指令序列是机器语言，用助记符号来表示机器指令便于记忆，这就形成了汇编语言。因此，用汇编语言写成的程序不能直接放入 MCU 的程序存储器中去执行，必须先转为机器语言。把用汇编语言写成的源程序“翻译”成机器语言的工具叫汇编程序或汇编器(Assembler)，以下统一称作汇编器。

本书给出的所有样例程序均在 KDS3.0 开发环境下实现，KDS3.0 环境默认使用 GNU 汇编器(Cross ARM GNU Assembler)，汇编语言格式满足 GNU 汇编语法，下面简称 ARM-GNU

汇编[①]。为了有助于解释涉及的汇编指令，下面将介绍一些汇编语法的基本信息。

2.4.1 ARM-GNU 汇编语言格式

汇编语言源程序可以用通用的文本编辑、软件编辑，以 ASCII 码形式存盘。具体的汇编器对汇编语言源程序的格式有一定的要求，同时，汇编器除了识别 MCU 的指令系统外，为了能够正确地产生目标代码以及方便汇编语言的编写，汇编器还提供了一些在汇编时使用的命令、操作符号，在编写汇编程序时，也必须正确使用它们。由于汇编器提供的指令仅是为了更好地做好“翻译”工作，并不产生具体的机器指令，因此这些指令被称为伪指令(Pseudo Instruction)。例如，伪指令告诉编译器：从哪里开始编译，到何处结束，汇编后的程序如何放置等相关信息。当然，这些相关信息必须包含在汇编源程序中，否则编译器就难以编译好源程序，难以生成正确的目标代码。

汇编语言源程序以行为单位进行设计，每一行最多可以包含以下几个部分：

```
标号:   操作码   操作数1,操作数2,…   注释
```

1. 标号

标号(Labels)可以确定代码当前位置的程序计数器(pc)值。对于标号有下列要求及说明。

(1) 如果一个语句有标号，则标号必须书写在汇编语句的开头部分。

(2) 标号可以由任何有效字符和冒号组成。有效字符包含以下字符：字母 A～Z、字母 a～z、数字 0～9、下画线“_”、美元符号“$”，但开头的第一个符号不能为数字和$。

(3) 汇编器对标号中字母的大小写敏感，但指令不区分大小写。

(4) 标号长度基本上不受限制，但实际使用时通常不要超过 20 个字符。若希望更多的汇编器能够识别，建议标号(或变量名)的长度小于 8 个字符。

(5) 标号后必须带冒号“:”。

(6) 一个标号在一个文件(程序)中只能定义一次，否则重复定义，不能通过编译。

(7) 一行语句只能有一个标号，汇编器将把当前程序计数器的值赋给该标号。

2. 操作码

操作码(Opcodes)包括指令码或伪指令，其中，伪指令是指 GNU 汇编器可以识别的伪指令。对于有标号的行，必须用至少一个空格或制表符(Tab)将标号与操作码隔开。对于没有标号的行，不能从第一列开始写操作码，应以空格或制表符(Tab)开头。汇编器不区分操作码中字母的大小写。

3. 操作数

操作数(Operands)可以是地址、标号或指令码定义的常数，也可以是由伪运算符构成的表达式。若一条指令或伪指令有操作数，则操作数与操作码之间必须用空格隔开书写。操作数多于一个的，操作数之间用逗号“,”分隔。操作数也可以是 CM0+内部寄存器，或者另一条指令的特定参数。操作数中一般都有一个存放结果的寄存器，这个寄存器在操作数

① Free Software Foundation Inc. Using as The GNU Assembler. Version 2.11.90.2012.

的最前面。

1）常数标识

汇编器识别的常数有十进制(默认不需要前缀标识)、十六进制(0x 前缀标识)、二进制(用 0b 前缀标识)。

2）“#”表示立即数

一个常数前添加“#”表示一个立即数，不加“#”时表示一个地址。

特别说明：初学者常常会将立即数前的“#”遗漏，如果该操作数只能是立即数时，汇编器会提示错误，例如：

```
mov  r3, 1      @给寄存器 r3 赋值为 1(这个语句不对)
```

编译时会提示“immediate expression requires a # prefix--'mov r3,1'”，应该改为：

```
mov  r3,#1      @给寄存器 r3 赋值为 1(这个语句对)
```

3）圆点“.”

若圆点“.”单独出现在语句的操作码之后的操作数位置上，则代表当前程序计数器的值被放置在圆点的位置。例如，b. 指令代表转向本身，相当于永久循环，在调试时希望程序停留在某个地方可以添加这种语句，调试之后应删除。

4）伪运算符

表 2-20 列出了一系列的伪运算符。

表 2-20　GNU 汇编器识别的伪运算符

<table>
<tr><th></th><th>运算符</th><th>功能</th><th>类型</th><th colspan="2">实　例</th></tr>
<tr><td rowspan="2">运算符前缀</td><td>-</td><td>负号</td><td>一元</td><td>ldr r3,=-325</td><td>等价于 ldr r3,=0xfffffebb</td></tr>
<tr><td>~</td><td>取反运算</td><td>一元</td><td>ldr r3,=~325</td><td>等价于 ldr r3,=0xfffffeba</td></tr>
<tr><td rowspan="19">优先级从大到小后缀运算符</td><td>*</td><td>乘法</td><td>二元</td><td>mov r3,#5*4</td><td>等价于 mov r3,#20</td></tr>
<tr><td>/</td><td>除法</td><td>二元</td><td>mov r3,#20/4</td><td>等价于 mov r3,#5</td></tr>
<tr><td>%</td><td>取模</td><td>二元</td><td>mov r3,#20%7</td><td>等价于 mov r3,#6</td></tr>
<tr><td><或<<</td><td>左移</td><td>二元</td><td>mov r3,#4<<2</td><td>等价于 mov r3,#16</td></tr>
<tr><td>>或>></td><td>右移</td><td>二元</td><td>mov r3,#4>>2</td><td>等价于 mov r3,#1</td></tr>
<tr><td>|</td><td>按位或</td><td>二元</td><td>mov r3,#4|2</td><td>等价于 mov r3,#6</td></tr>
<tr><td>&</td><td>按位与</td><td>二元</td><td>mov r3,#4 ^2</td><td>等价于 mov r3,#0</td></tr>
<tr><td>^</td><td>按位异或</td><td>二元</td><td>mov r3,#4 ^6</td><td>等价于 mov r3,#2</td></tr>
<tr><td>!</td><td>逻辑非</td><td>二元</td><td>mov r3,! #1</td><td>等价于 mov r3,#0</td></tr>
<tr><td>+</td><td>加法</td><td>二元</td><td>mov r3,#30+40</td><td>等价于 mov r3,#70</td></tr>
<tr><td>-</td><td>减法</td><td>二元</td><td>mov r3,#40-30</td><td>等价于 mov r3,#10</td></tr>
<tr><td>==</td><td>等于</td><td>二元</td><td>mov r3,#1==0</td><td>等价于 mov r3,#0</td></tr>
<tr><td><></td><td>不等于</td><td>二元</td><td>mov r3,#1<>1</td><td>等价于 mov r3,#0</td></tr>
<tr><td>></td><td>大于</td><td>二元</td><td>mov r3,#1>0</td><td>等价于 mov r3,#1</td></tr>
<tr><td><</td><td>小于</td><td>二元</td><td>mov r3,#1<0</td><td>等价于 mov r3,#0</td></tr>
<tr><td>>=</td><td>大于等于</td><td>二元</td><td>mov r3,#1>=0</td><td>等价于 mov r3,#1</td></tr>
<tr><td><=</td><td>小于等于</td><td>二元</td><td>mov r3,#1<=0</td><td>等价于 mov r3,#0</td></tr>
<tr><td>&&</td><td>逻辑与</td><td>二元</td><td>mov r3,#1&&0</td><td>等价于 mov r3,#0</td></tr>
<tr><td>||</td><td>逻辑或</td><td>二元</td><td>mov r3,#1||0</td><td>等价于 mov r3,#1</td></tr>
</table>

4. 注释

注释(Comments)即是说明文字,类似于C语言,多行注释以“/*”开始,以“*/”结束。这种注释可以包含多行,也可以独占一行。在GNU汇编器的汇编语言中,单行注释以“@”引导或者用“#”引导,用@引导类似于C语言的“//”。用“#”引导时,“#”必须为单行的第一个字符。

2.4.2 伪指令

为了方便汇编语言的编写以及编译器能够正确地产生目标代码,编译器还提供了一些伪指令。所谓伪指令就是没有对应的机器码的指令,它是用于告诉编译器如何进行汇编的指令,它既不控制机器的操作也不被汇编成机器代码,只能为编译器所识别并指导汇编如何进行。这些伪指令主要有用于常量以及宏的定义、条件判断、文件包含等伪指令。所有的汇编命令都是以“.”开头。

1. 系统预定义的段

在KDS3.0开发环境下,C语言程序在经过gcc编译器后最终生成.elf格式的可执行文件。.elf可执行程序是以段为单位来组织文件的。通常划分为如下几个段:.text、.data和.bss,其中,.text是只读的代码区,.data是可读可写的数据区,而.bss则是可读可写且没有初始化的数据区。.text段开始地址为0x0,接着分别是.data段和.bss段。

```
.text        @表明以下代码在.text段
.data        @表明以下代码在.data段
.bss         @表明以下代码在.bss段
```

2. 常量的定义

汇编代码常用的功能之一为常量的定义。使用常量定义,能够提高程序代码的可读性,并且使代码维护更加简单。常量的定义可以使用.equ汇编指令,下面是GNU汇编器的一个常量定义的例子:

```
.equ  _NVIC_ICER, 0xE000E180
...
LDR  R0,=_NVIC_ICER                @将0xE000E180放到R0中
```

常量的定义还可以使用.set汇编指令,其语法结构与.equ相同。

```
.set  ROM_size, 128 * 1024              @ROM大小为131072字节(128KB)
.set  start_ROM, 0xE0000000
.set  end_ROM, start_ROM + ROMsize      @ROM结束地址为0xE0020000
```

3. 程序中插入常量

对于大多数汇编工具来说,一个典型特性是可以在程序中插入数据。GNU汇编器语法可以写作:

```
LDR R3, =NUMNER               @得到 NUMNER 的存储地址
LDR R4, [R3]                  @将 0x123456789 读到 R4
    ...
    LDR R0, =HELLO_TEXT       @得到 HELLO_TEXT 的起始地址
    BL PrintText              @调用 PrintText 函数显示字符串
    ...
    ALIGN 4
NUMNER:
    .word 0x123456789
HELLO_TEXT:
    .asciz "hello\n"          @以'\0'结束的字符
```

为了在程序中插入不同类型的常量,GNU 汇编器中包含许多不同的伪指令,表 2-21 中列出了常用的例子。

表 2-21　用于程序中插入不同类型常量的常用伪指令

插入数据的类型	GNU 汇编器
字	.word(例如,.word 0x123456789)
半字	.hword(例如,.word 0x12345)
字节	.byte(例如,.byte 0x12)
字符串	ascii/.asciz(例如,.ascii "hello\n",.asciz 与.ascii,只是生成的字符串以'\0'结尾)

4. 条件伪指令

.if 条件伪指令后面紧跟着一个恒定的表达式(即该表达式的值为真),并且最后要以.endif 结尾。中间如果有其他条件,可以用.else 填写汇编语句。

.ifdef 标号,表示如果标号被定义,执行下面的代码。

5. 文件包含伪指令

```
.include "filename"
```

.include 是一个附加文件的链接指示命令,利用它可以把另一个源文件插入当前的源文件一起汇编,成为一个完整的源程序。filename 是一个文件名,可以包含文件的绝对路径或相对路径,但建议对于一个工程的相关文件放到同一个文件夹中,所以更多的时候使用相对路径。具体例子可参见第 3 章的第一个汇编实例程序。

6. 其他常用伪指令

除了上述的伪指令外,GNU 汇编还有其他常用伪指令。

(1) .section 伪指令:用户可以通过.section 伪指令来自定义一个段。例如:

```
.section   .isr_vector,   "a"        @定义一个.isr_vector 段,"a"表示允许段
```

(2) .global 伪指令:.global 伪指令可以用来定义一个全局符号。例如:

```
.global   symbol      @定义一个全局符号 symbol
```

(3) .extern 伪指令：.extern 伪指令的语法为：.extern symbol，声明 symbol 为外部函数，调用的时候可以遍访所有文件找到该函数并且使用它。例如：

```
.extern   main        @声明 main 为外部函数
bl        main        @进入 main 函数
```

(4) .align 伪指令：.align 伪指令可以通过添加填充字节使当前位置满足一定的对齐方式。语法结构为：.align [exp[, fill]]，其中，exp 为 0～16 之间的数字，表示下一条指令对齐至 2^{exp} 位置，若未指定，则将当前位置对齐到下一个字的位置，fill 给出为对齐而填充的字节值，可省略，默认为 0x00。例如：

```
.align   3     @把当前位置计数器值增加到 2^3 的倍数上,若已是 2^3 的倍数,不做改变
```

(5) .end 伪指令：.end 伪指令声明汇编文件的结束。

还有有限循环伪指令、宏定义和宏调用伪指令等，参见网上教学资源中《嵌入式技术基础与实践(第 4 版)》补充阅读材料及 GNU 汇编语法。

小　　结

本章简要介绍了 CM0+的内部寄存器、指令系统及汇编伪指令，有助于读者更深层次地理解和学习 CM0+软硬件的设计。

(1) 2012 年推出的 CM0+处理器基于 ARMv6M 架构设计，M0+处理器主要包含 M0+内核、嵌套中断向量控制器(NVIC)、总线网络(BusMatrix)、调试组件、总线接口、SysTick 定时器及其他控制模块。ARM 公司给出了 M0+的 4GB 空间的大致用途，供芯片制造商设计实际芯片时参考。M0+处理器内含 13 个通用寄存器 R0～R12、堆栈指针 R13(SP)、连接寄存器 R14(LR)、程序计数寄存器 R15(PC)、程序状态字寄存器组 xPSR、中断屏蔽寄存器 PRIMASK、控制寄存器 CONTROL。通用寄存器 R0～R12 中的 R0～R7 被称为低组寄存器，所有指令都能访问它们。R8～R12 被称为高组寄存器。只有很少的 16 位 Thumb 指令能访问它们，32 位的 Thumb2 指令则不受限制。

(2) M0+采用精简指令集 RISC，一共只有 57 条基本指令。M0+的寻址方式有立即数寻址、偏移寻址及寄存器间接寻址、直接寻址。这 57 条基本指令依据不同寻址方式形成 68 条具体指令，归纳起来分为数据传送大类、数据操作类、跳转控制类和其他指令这 4 大类。数据传送类指令的功能有两种情况，一是取存储器地址空间中的数传送到寄存器中，二是将寄存器中的数传送到另一寄存器或存储器地址空间中，典型的有 LDR、STR、MOV、PUSH、POP 等；数据操作主要指算术运算、逻辑运算、移位等，如加 ADD、减 SUB、乘 MUL、逻辑与 AND、或 ORR、异或 EOR 等；跳转类指令用来控制程序的执行流程，如 B、BL、BX、BLX 等；其他指令主要用到开总中断 CPSIE i 和关总中断 CPSID i 两条。

(3) 2.3 节给出了 CM0+指令集与机器码对应表，目的是在一些深入细致的调试分析中，需要分析机器码，有了这张表便于进行这项工作。初学者了解即可。从指令集与机器码

对应表中也了解到，M0+大部分指令为 16 位，只有极少部分为 32 位。

(4) 2.4 节给出了 GNU 下的汇编语言书写格式及一些伪指令，学习这些内容及汇编语言指令对理解后面章节的汇编工程来说是必需的，虽然汇编语言在实际编程中已经较少用到，但涉及对效率要求极高的情况下时，还是需要使用汇编进行设计。此外，彻底理解一个汇编工程并完成一个汇编工程，对打好嵌入式学习功底极有益处。

习　题

1. ARM Cortex-M0+处理器有哪些寄存器？简述各个寄存器的作用。
2. 说明对 CPU 内部寄存器的操作与对 RAM 中的全局变量操作有何异同点。
3. ARM Cortex-M0+指令系统寻址方式有几种？简要叙述各自特点，并举例说明。
4. 调用子程序是用 B 还是用 BL 指令？请写出返回子程序的指令。
5. 举例说明运算指令与伪运算符的本质区别。

第 3 章 存储映像、中断源与硬件最小系统

本章导读：本章简要概述了 KL25/26 的存储映像、中断源与硬件最小系统，有助于读者了解 KL25/26 软硬件系统的大致框架，以便开始 KL25/26 的软硬件设计。3.1 节简要介绍了 Kinetis 全系列微控制器产品分类及应用领域；3.2 节给出 KL 系列 MCU 的型号标识、基本特点及体系结构概述；3.3 节给出 KL25/26 系列芯片的存储映像及中断源，存储映像主要包括 Flash 区、片内 RAM 区，以便于配置链接文件；3.4 节将引脚分为硬件最小系统引脚及对外提供服务的引脚两类，并介绍其各自的功能；3.5 节给出了 KL25/26 硬件最小系统的原理图及简明分析。

本章参考资料：3.1 节主要参考自官网 Kinetis 的简介；3.2 节主要参考自官网资料 KL 系列介绍以及《KL 参考手册》①；3.3 节的 KL25/26 系列存储映像与中断源参考自《KL 参考手册》的第 3 章；3.4 节的 KL25/26 的引脚功能，参考自《KL 参考手册》的第 10 章。

3.1 恩智浦 Kinetis 系列微控制器简介

飞思卡尔（2015 年与恩智浦合并）在 2010 年飞思卡尔技术论坛（FTF2010）美国站推出了 Kinetis 系列微控制器，这是基于新 ARM Cortex-M4 处理器的 90nm 32 位 MCU，开创了其微控制器领先地位的新纪元。它基于低功率混合信号 ARM Cortex-M4 处理器，是业内扩展能力最强的 MCU 系列之一。面向不同应用领域，Kinetis 系列基于不同的 ARM Cortex-M 内核陆续推出了 Kinetis K 系列、L 系列、M 系列、W 系列、E 系列、EA 系列以及 V 系列。了解这些系列概况有助于应用时选型。

1. K 系列

Kinetis K 系列产品组合有超过两百种基于 ARM Cortex-M4 结构的低功耗、高性能、可兼容的微控制器。这个系列产品集成度高，它包含多种快速 16 位 ADC、DAC 和可编程增益放大器以及强大、经济有效的信号转换器。目标应用领域是便携式医疗设备、仪器仪表、工业控制及测量设备等。

2. KL 系列

Kinetis L 系列（KL 系列）MCU 不仅汲取了新型 ARM Cortex-M0＋处理器的卓越能效和易用性、功耗更低、价格更低、效率更高，而且体现了 Kinetis 产品优质的性能、多元化的外设、广泛的支持和可扩展性。目标应用领域是 8/16 位 MCU 应用领域的升级换代，适用于价格敏感、能效比相对较高的领域，如手持设备、智能终端等。

3. KM 系列

KM 系列也是基于 32 位 ARM Cortex-M0＋内核的 MCU。所有 KM 系列 MCU 都包

① 本书随后所说的《KL 参考手册》均分别指《KL25 参考手册》《KL26 参考手册》。

含一个模拟前端，使 CPU 的电源计算可以达到 0.1%的精确度。它包含 4 个 24 位 Σ-Δ 模数转换器、两个低噪声可编程增益放大器，温度漂移范围小和具有相对补偿的精密参考电压，以简化精确的功率计算。目标应用领域是经济高效的单相或两相电表设计中。

4. KW 系列

KW 系列 MCU 扩展了 K 系列基于 ARM Cortex-M4 的成功之处。KW20 无线 MCU 集成了领先的 RF 收发器和 ARM Cortex-M4 内核，并且支持一个强大的、安全的、可靠的和低功耗的 IEEE 802.15.4 的无线解决方案。KW 系列是一个性能优越的无线 MCU 选择方案，可以提供良好的混合性能、集成、连通性和安全性。KW01 超低功耗无线 MCU 是基于 ARM Cortex-M0＋内核的智能无线解决方案，旨在解决低于 1GHz（290～1020MHz）的无线连接应用。目标应用领域是智能电表、传感器控制网络、工业控制、数据采集等。

5. KE 系列

KE 系列产品可在复杂电气噪声环境和要求高可靠性的应用中保持高稳定性，而且有丰富的存储器、外设和产品包可供选择。它们具有通用的外设和引脚数量，使开发人员能够轻松实现相同 MCU 系列内或多个 MCU 系列间的迁移，以利用更多存储器或特性集成。这种可扩展性使开发人员能够在 KE 系列上实现其终端产品平台的标准化，最大程度地提高硬件和软件的再利用，并加快产品上市速度。

6. KEA 系列

KEA 系列 32 位 MCU 广泛适用于质量要求和长期供货保证要求都很高的汽车和工业应用。KEA 系列是广泛的 ARM 生态合作体系的入门级产品，拥有出色的 EMC/ESD 兼容性，能够适应高温环境，并且辐射排放较低。恩智浦为 KEA 系列提供可扩展、稳定可靠的高性能解决方案，适合成本敏感型汽车应用。此外恩智浦还提供了丰富的参考设计、工具和应用说明，最大程度缩短设计开发时间，加快产品上市速度。

7. KV 系列

KV 系列 MCU 基于 ARM Cortex-M0＋、Cortex-M4 和 Cortex-M7 内核，专为各种 BLDC、PMSM 和 ACIM 电机控制以及数字电源转换应用而设计。凭借飞思卡尔逾二十年的电机控制处理器专长，推出针对电机控制和数字功率的 MCU 是顺理成章的事。KV 系列凭借出色的性能/价格比、量身定制的外设和捆绑型电机套件，可以使开发人员比以前更快、更容易地进行高效的设计。

3.2　KL 系列 MCU 简介与体系结构概述

3.2.1　KL 系列 MCU 简介

KL 系列 MCU 于 2012 年 6 月提供样片，2013 年正式上市。该系列 MCU 是业内首款基于 ARM Cortex-M0＋内核的 MCU，具有超低功耗、应用设计方便、扩展性好、系列品种齐全等特点。目标市场是传统 8 位 MCU 应用领域的 32 位升级换代。Kinetis L 系列 MCU 面向家用电器、便携式医疗系统、智能电表、照明、电源、电机控制及工业控制系统等，对稳定性、功耗、成本和易用性等方面有较严格要求的市场。该系列的设计充分考虑了应用

的简约性,使得应用工程师可以像使用8位机一样进行32位机的应用开发。同时,KL系列MCU和基于ARM Cortex-M4内核的K系列MCU完全兼容,包括引脚。为提高性能、扩大闪存和未来集成提供了扩展途径,也使得软件、硬件的可复用性与可移植性得到了较宽的延伸。

KL系列MCU具有多个低功率操作模式,包括新的门控时钟,该模式在要求最低功耗时通过关闭总线、系统时钟减少动态功耗,外设仍可在一个可选异步时钟源下继续运作;在未唤醒内核情况下,UART、SPI、I2C、ADC、DAC、TPM、LPT和DMA等可支持低功耗模式。KL系列MCU主要特点如下:CPU最高工作频率48MHz、支持直接存储器访问(Direct Memory Access,DMA),位操作引擎(Bit Manipulation Engine,BME)、内核单周期访问内存速度可达1.77CoreMark/MHz、单周期访问I/O、关键外设速度比标准I/O最大提高50%;2级流水线设计减少了指令周期数(CPI),提高跳转指令和执行ISR中断服务例程速度;与8位、16位MCU相比具有更精简的代码密度,减少Flash空间、系统资源及功耗;更精简的指令系统优化访问程序存储空间,完全兼容ARM Cortex-M0,兼容Cortex-M3/M4指令集子集。执行跟踪缓冲区:实现轻量级追踪解决方案,更快定位修正bug。

1. KL系列MCU的型号标识

恩智浦Kinetis系列MCU的型号众多,但同一子系列的CPU核是相同的,多种型号只是为了适用于不同的应用场合。为了方便选型或订购,需记忆MCU型号标识的基本含义。KL系列命名格式为:“**Q　KL##　A　FFF　R　T　PP　CC　(N)**”,其中,各字段说明如表3-1所示,本书使用的芯片命名为MKL25Z128VLK4。

表3-1　KL系列芯片命令字段说明

字　段	说　明	取　值
Q	质量状态	M=正式发布芯片;P=工程测试芯片
KL##	Kinetis系列号	KL25
A	内核属性	Z=Cortex-M0+
FFF	程序Flash大小	32=32KB;64=64KB;128=128KB;256=256KB
R	硅材料版本	(空)=主要使用的版本;A=主要使用版本的更新
T	运行温度范围	V=−40~105℃
PP	封装类型	FM=32 QFN(5mm×5mm);FT=48 QFN(7mm×7mm);LH=64 LQFP(10mm×10mm);LK=80 LQFP(12mm×12mm)
CC	CPU最高频率	4=48MHz
N	包装类型	R=卷包装;(空)=盒包装

2. KL系列MCU的共性

KL系列MCU由5个子系列组成,分别是KL0x、KL1x、KL2x、KL3x、KL4x,表3-2给出了KL系列芯片的简明资源。所有KL系列MCU均具有低功耗与丰富的混合信号控制外设,提供了不同的闪存容量和引脚数量,供实际应用选型。从应用的角度而言,KL0x属于入门级芯片,KL1x属于通用型芯片,而KL2x、KL3x、KL4x则更具针对性,KL2x系列具有USB OTG技术,KL3x系列支持段式LCD,KL4x系列为KL的旗舰系列,支持功能也最丰富。

表 3-2　KL 系列芯片的简明资源

系列	Flash 大小	引脚数量	低功耗	模拟信号	USB	段式 LCD	备注
KL4x	128～256KB	64～121	√	√	√	√	
KL3x	64～256KB	64～121	√	√		√	
KL2x	32～256KB	32～121	√	√	√		本书选用
KL1x	32～256KB	32～80	√	√			
KL0x	8～32KB	16～48	√	√			

KL 系列 MCU 在内核、低功耗、存储器、模拟信号、人机接口、安全性、定时器及系统特性等方面具有一些共同特点，简要总结在表 3-3 中。

表 3-3　KL 系列 MCU 的共性

项　　目	特　　点
超低功耗	32 位 ARM Cortex-M0＋内核具有超低功耗
内存	可扩展内存：8KB Flash/1KB SRAM 至 128KB Flash/16KB SRAM；内含 64B 高速缓冲存储区，可优化总线宽度和 Flash 的执行性能(KL02 系列除外)
模拟信号	快速、高精度 16/12 位 ADC；12 位 DAC；高速比较器
人机接口	低功率触摸感应界面
通信	所有 UART 支持 DMA 传输，总线检测到数据也能触发传输；UART0 支持 4～32 倍的采样速率；在 STOP/VLPS 模式，也能运行异步传输和接收操作；最大支持两路 SPI；最大支持两路 I2C；支持全速 USB OTG 片内传输控制设备
可靠性、安全性	内部看门狗监控
定时控制器	强大的定时模块支持通用/PWM/电机控制功能；可用于 RTOS 任务调度、ADC 转换或定时的周期中断定时器
系统特性	GPIO 支持引脚中断；宽泛的工作电压：1.71～ 3.6V；Flash 编程电压、模拟外设电压低至 1.71V；运行温度范围：－40℃～ 105℃

3. KL25/26 子系列 MCU 简明资源

本书以 KL25 与 KL26 子系列为蓝本阐述嵌入式技术基础，至本书出版时，该系列各共有 12 个具体芯片型号。共同特点有：CPU 工作频率为 48MHz；工作电压为 1.71～ 3.6V；运行温度范围为－40℃～105℃；具有 64B 的 Cache；具有 USB OTG、定时器、DMA、UART、SPI、I2C、TSI、16 位 ADC、12 位 DAC 等模块。在 Flash 容量、RAM 容量、I/O 引脚数及封装形式等有差异，见表 3-4，带底纹的型号为本书选用。具体应用时，需查询芯片的数据手册。

表 3-4　KL25/26 子系列 MCU 简明资源

引脚数	封装	Flash/KB	SRAM/KB	KL25 型号	KL26 型号
32	QFN	32	4	MKL25Z32VFM4(R)	MKL26Z32VFM4(R)
		64	8	MKL25Z64VFM4(R)	MKL26Z64VFM4(R)
		128	16	MKL25Z128VFM4(R)	MKL26Z128VFM4(R)
48	QFN	32	4	MKL25Z32VFT4(R)	MKL26Z32VFT4(R)
		64	8	MKL25Z64VFT4(R)	MKL26Z64VFT4(R)
		128	16	MKL25Z128VFT4(R)	MKL26Z128VFT4(R)

续表

引脚数	封装	Flash/KB	SRAM/KB	KL25 型号	KL26 型号
64	LQFP	32	4	MKL25Z32VLH4(R)	MKL26Z32VLH4(R)
		64	8	MKL25Z64VLH4(R)	MKL26Z64VLH4(R)
		128	16	MKL25Z128VLH4(R)	MKL26Z128VLH4(R)
80	LQFP	32	4	MKL25Z32VLK4(R)	MKL26Z32VLK4(R)
		64	8	MKL25Z64VLK4(R)	MKL26Z64VLK4(R)
		128	16	MKL25Z128VLK4(R)	MKL26Z128VLK4(R)

3.2.2 KL 系列 MCU 体系结构概述

KL 系列 MCU 是以 AMBA 总线规范为架构的片上系统(System On Chip,SOC),如图 3-1 所示。一般来说,AMBA 架构包含高性能系统总线(Advanced High Performance Bus,AHB)和低速、低功耗的高级外设总线(Advanced PeriPheral Bus,APB)。高性能系统总线 AHB 是负责连接 ARM 内核、DMA 控制器、片内存储器或其他需要高带宽的模块。而外设总线 APB 则是用来连接系统的外围慢速模块,其协议规则相对系统总线 AHB 来说较为简单,它与系统总线 AHB 之间则通过总线桥(Bus Bridge)相连,期望能减少系统总线的负载。

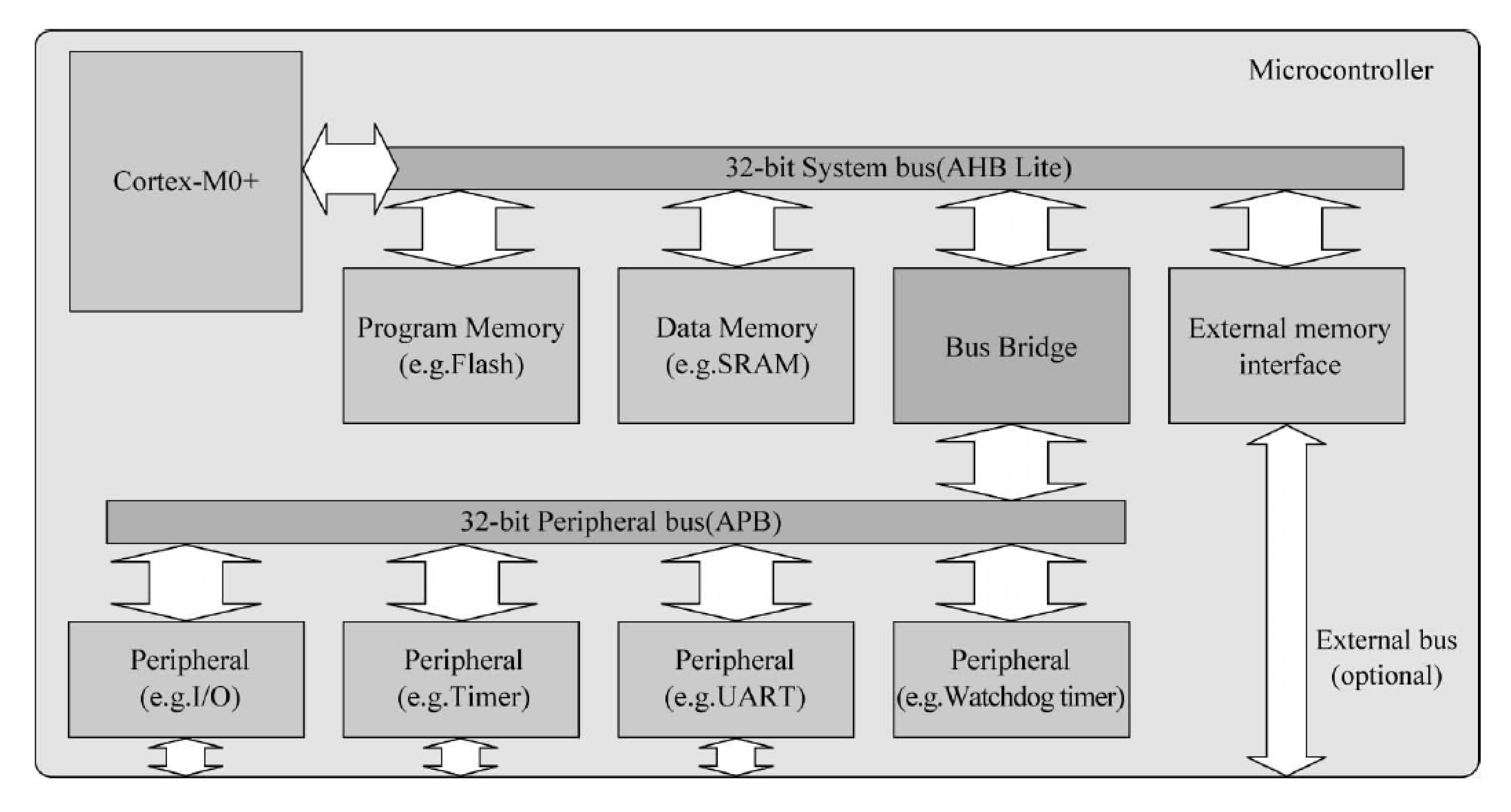


图 3-1 32 位 MCU 系统、外围总线模块图

1. AMBA 总线规范

ARM 公司定义了 AMBA(Advanced Microcontroller Bus Architecture)总线规范,它是一组针对基于 ARM 内核、片内系统之间通信而设计的标准协议。在 AMBA 总线规范中,定义三种总线,分别是:①高性能总线,用于高性能系统模块的连接,支持突发模式数据传输和事务分割;②高级系统总线(Advanced System Bus,ASB),也用于高性能系统模块的连接,支持突发模式数据传输,这是较老的系统总线格式,后来由高性能总线 AHB 替代;

③高级外设总线(Advanced PeriPheral Bus,APB),用于较低性能外设的简单连接,一般是接在 AHB 或 ASB 系统总线上的第二级总线。最初的 AMBA 总线是 ASB 和 APB。在它的第二个版本中,ARM 引入了 AHB。

2. 总线桥

总线桥(Bus Bridge),在《KL 参考手册》中也被称为外设桥(Peripheral Bridge),给外设桥的名字是 AIPS-Lite,即图 3-1 中的 Bus Bridge。外设桥的作用是把交叉开关(Crossbar Switch)接口协议,转换成私有外设总线协议(IPS/APB)。本书中 MCU 外设桥以外设槽(Slot)形式,最多可连接 128 个外设,给每个外设分配 4KB 的寄存器映像空间。外设桥为每个外设槽提供了独立时钟,以便更好地支持低速外设。

3. 交叉开关

交叉开关(Crossbar Switch)将总线主机与总线从机相连,该结构允许多达 4 路主机同时访问不同总线从机。所谓总线主机是指其可在总线上产生与控制所有时序。当总线主机试图通过交叉开关访问从机端口,若该端口处于空闲状态,则执行访问,访问速度最大可达单周期或零等待;若从机端口处于忙状态或被其他总线主机占用,总线主机插入查询等待状态直到目的从机响应服务主机请求。响应服务延迟时间取决于每个主机各自优先级以及访问目的从机时间。但若同时访问相同从机时,交叉开关提供仲裁机制确定访问顺序,仲裁机制包括优先级固定算法和优先级轮转算法。

3.3　KL25/26 系列存储映像与中断源

3.3.1　KL25/26 系列存储映像

所谓存储映像(Memory Mapping)在这里可以直观地理解为,M0+寻址的 4GB 地址空间(0x0000_0000～0xFFFF_FFFF)[①]被如何使用,都对应了哪些实际的物理介质。有的给了 Flash 存储器使用,有的给了 RAM 使用,有的给了外设模块使用。下面利用 GPIO 模块来阐述有关概念。GPIO 模块使用了 0x400F_F000～0x400F_FFFF 地址空间,这些空间内的 GPIO 寄存器与 CPU(即 M0+内核)内部寄存器(如 R0、R1 等)不同,访问 GPIO 寄存器需要使用直接地址进行访问,也就是说需要使用三总线(地址总线、数据总线、控制总线)。而访问 CPU 内部寄存器,不需经过三总线(汇编语言直接使用 R0、R1 等名称即可),没有地址问题。由于访问 CPU 内部寄存器不经过三总线,所以比访问 GPIO 寄存器(对应直接地址)来得快。为区别于 CPU 内部寄存器,GPIO 寄存器也被称为“映像寄存器”(Mapping Register),相对应的地址被称为“映像地址”(Mapping Address)。整个可直接寻址的空间被称为“映像地址空间”(Mapping Address Space)。

KL25/26 把 M0+内核之外的模块,用类似存储器编址的方式,统一分配地址。在 4GB 的映像地址空间内,分布着片内 Flash、SRAM、系统配置寄存器以及其他外设等,以便 CPU 通过直接地址进行访问。表 3-5 给出了本书中介绍的 MKL25Z128VLK4(简称 KL25,下

---

① 0x00000000 书写成 0x0000_0000 仅仅是为了清晰,便于阅读。

同)和MKL26Z128VLH4(简称KL26,下同)存储映像空间分配。

表3-5　KL25/26存储映像空间分配

32位地址范围	目的从机	说明
0x0000_0000～0x07FF_FFFF	可编程Flash和只读数据	128KB(0x0000_0000～0x0001 FFFF)
0x0800_0000～0x1FFF_EFFF	保留	—
0x1FFF_F000～0x1FFF_FFFF	SRAM_L：Lower SRAM	16KB RAM区
0x2000_0000～0x2000_2FFF	SRAM_U：Upper SRAM	
0x2000_3000～0x3FFF_FFFF	保留	—
0x4000_0000～0x4007_FFFF	AIPS外围设备	串口、定时器、模块配置等
0x4008_0000～0x400F_EFFF	保留	—
0x400F_F000～0x400F_FFFF	通用输入/输出(GPIO)	GPIO模块
0x4010_0000～0x43FF_FFFF	保留	—
0x4400_0000～0x5FFF_FFFF	BME访问外设槽0-127	只能对特定的区域位操作
0x6000_0000～0xDFFF_FFFF	保留	—
0xE000_0000～0xE00F_FFFF	私有外设	系统时钟、中断控制器、调试接口
0xE010_0000～0xEFFF_FFFF	保留	～
0xF000_0000～0xF000_0FFF	MTB(微型跟踪缓存)寄存器	—
0xF000_1000～0xF000_1FFF	MTB数据查看和跟踪寄存器	—
0xF000_2000～0xF000_2FFF	ROM表	存放存储映射信息
0xF000_3000～0xF000_3FFF	杂项控制模块	—
0xF000_4000～0xF7FF_FFFF	保留	—
0xF800_0000～0xFFFF_FFFF	IOPORT：GPIO(单周期访问)	可被内核直接访问

对于此表,主要记住片内Flash区及片内RAM区存储映像。因为中断向量、程序代码、常数放在片内Flash中,源程序编译后的链接阶段需要使用的链接文件中需含有目标芯片Flash的地址范围及用途等信息,才能顺利生成机器码。链接文件中还需包含RAM的地址范围及用途等信息,以便生成机器码确切定位全局变量、静态变量的地址及堆栈指针。

1. 片内Flash区存储映像

KL25/26片内Flash大小为128KB,地址范围是：0x0000_0000 ～ 0x0001_FFFF,一般被用来存放中断向量、程序代码、常数等,其中前192B为中断向量表。

2. 片内RAM区存储映像

KL25/26片内RAM为静态随机存储器SRAM,大小为16KB,地址范围是：0x1FFF_F000～0x2000_2FFF,一般被用来存储全局变量、静态变量、临时变量(堆栈空间)等。这16KB的RAM,在物理上被划分为SRAM_L和SRAM_U两个部分①,分为SRAM_L：0x1FFF_F000～0x1FFF_FFFF(4KB)；SRAM_U：0x2000_0000～0x2000_2FFF(12KB)。该芯片的堆栈空间的使用方向是向小地址方向进行的,因此,堆栈的栈顶应该设置为RAM地址的最大值+1。这样,全局变量及静态变量从RAM的最小地址向大地址方向开始使用,堆栈从RAM的最高地址向小地址方向使用,可以减少重叠错误。

① 将SRAM划分为SRAM_L和SRAM_U。SRAM_U不仅可以作为普通RAM来操作,还可以支持两种途径的位操作,分别是位带别名区、位操作引擎(BME),供特殊功能下高级编程使用。

3. 其他存储映像

其他存储映像，如外设区存储映像（外设桥、GPIO、位操作引擎等）、私有外设总线存储映像、系统保留段存储映像等，只需了解即可，实际使用时，由芯片头文件给出宏定义。需特殊说明的是位操作引擎 BME：支持 BME 位操作引擎存储区地址位于 0x4400_0000～0x5FFF_FFFF。用于对外设的位操作，位操作引擎技术由硬件支持，可使用 Cortex-M 指令集中最基本的加载、存储指令完成对外设地址空间内存的读、改、写操作。具体用法见 13.4 节。

3.3.2 KL25/26 中断源

中断是计算机发展中一个重要的技术，它的出现很大程度上解放了处理器，提高了处理器的执行效率。所谓中断，是指 MCU 在正常运行程序时，由于 MCU 内核异常或者 MCU 各模块发出请求事件，引起 MCU 停止正在运行的程序，而转去处理异常或执行处理外部事件的程序（又称中断服务程序）。

这些引起 MCU 中断的事件称为中断源。KL25/26 的中断源分为两类，如表 3-6 所示，一类是内核中断，另一类是非内核中断。内核中断主要是异常中断，也就是说，当出现错误的时候，这些中断会复位芯片或是做出其他处理。非内核中断是指 MCU 各个模块中断源引起的中断，MCU 执行完中断服务程序后，又回到刚才正在执行的程序，从停止的位置继续执行后续的指令。非内核中断又称可屏蔽中断，这类中断可以通过编程控制，开启或关闭该中断。

表 3-6　KL25/26 的中断源

中断类型	中断向量号	IRQ 中断号	IPR 寄存器号	中断源	中断源说明
内核中断	0～3			ARM 内核	
	4～10			预留	
	11			ARM 内核	
	12,13			预留	
	14,15			ARM 内核	
非内核中断	16～19	0～3	0	DMA	DMA 通道 0～3 传输完成或错误
	20	4	1	预留	
	21	5	1	FTFA	命令完成或者读冲突
	22	6	1	PMC	低电压检测和警告中断
	23	7	1	LLWU	低漏唤醒
	24,25	8,9	2	I2C0,I2C1	I2C0,I2C1 中断
	26,27	10,11	2	SPI0,SPI1	SPI0,SPI1 中断
	28～30	12～14	3	UART0～2	UART0～2 状态和错误中断
	31	15	3	ADC0	ADC 转换完成中断
	32	16	4	ACMP0	ACMP 中断
	33～35	17～19	4	TPM0～2	TPM0～2 中断
	36	20	5	RTC	RTC 定时报警中断
	37	21	5	RTC	RTC 秒中断
	38	22	5	PIT	PIT 中断

续表

中断类型	中断向量号	IRQ中断号	IPR寄存器号	中　断　源	中断源说明
非内核中断	39	23	5	I2S0	KL25中无I2S模块
	40	24	6	USBOTG	
	41	25	6	DAC0	
	42	26	6	TSI0	
	43	27	6	MCG	
	44	28	7	LPTMR0	
	45	29	7	预留	
	46	30	7	端口控制模块	端口A引脚检测
	47	31	7	端口控制模块	端口C,D引脚检测

表3-6中还给出了各中断源的中断向量序号,非内核中断的中断请求(Interrupt Request)号(简称IRQ中断号),以及非内核中断的优先级设置的寄存器号(简称IPR寄存器号)。中断向量序号是每一个中断源的固定编号,由芯片设计生产时决定,编程时不能更改,它代表了中断服务程序入口地址在中断向量表的位置。IRQ中断号是非内核中断源的编号,每一个编号代表一个非内核中断源。6.3节将讲述中断的基本编程方法。

3.4　KL25/26的引脚功能

本书以80引脚LQFP封装的MKL25Z128VLK4芯片与64引脚LQFP封装的MKL26Z128VLH4芯片为例阐述ARM Cortex-M0+架构的Kinetis MCU的编程和应用。若没有特殊说明,本书的KL25均指MKL25Z128VLK4芯片,KL26均指MKL26Z128VLH4芯片。图3-2给出了80引脚LQFP封装的MKL25Z128VLK4的引脚图①,图3-3给出的是64引脚LQFP封装的MKL26Z128VLH4的引脚图②。

每个引脚都可能有多个复用功能,有的引脚有两个复用功能,有的有4个复用功能,实际嵌入式产品的硬件系统设计时必须注意只能使用其中的一个功能。进行硬件最小系统设计时,一般以引脚的第一功能作为引脚名进行原理图设计,若实际使用的是其另一功能,可以用括号加以标注,这样设计的硬件最小系统就比较通用。

下面从需求与供给的角度把MCU的引脚分为"硬件最小系统引脚"与"I/O端口资源类引脚"两大类。

3.4.1　硬件最小系统引脚

KL25/26硬件最小系统引脚是我们需要为芯片提供服务的引脚,包括电源类引脚、复位引脚、晶振引脚等,表3-7中给出了KL25/26的最小系统引脚。KL25/26芯片电源类引

① 来自《KL25参考手册》第10章图10-2,该章还给出了KL25的64引脚LQFP封装及48引脚QFN封装的引脚图。

② 来自《KL26参考手册》第10章图10-2,该章还给出了KL26的121引脚BGA封装、100引脚LQFP封装、64引脚MAPBGA封装、48引脚QFN封装及32引脚QFN封装的引脚图。

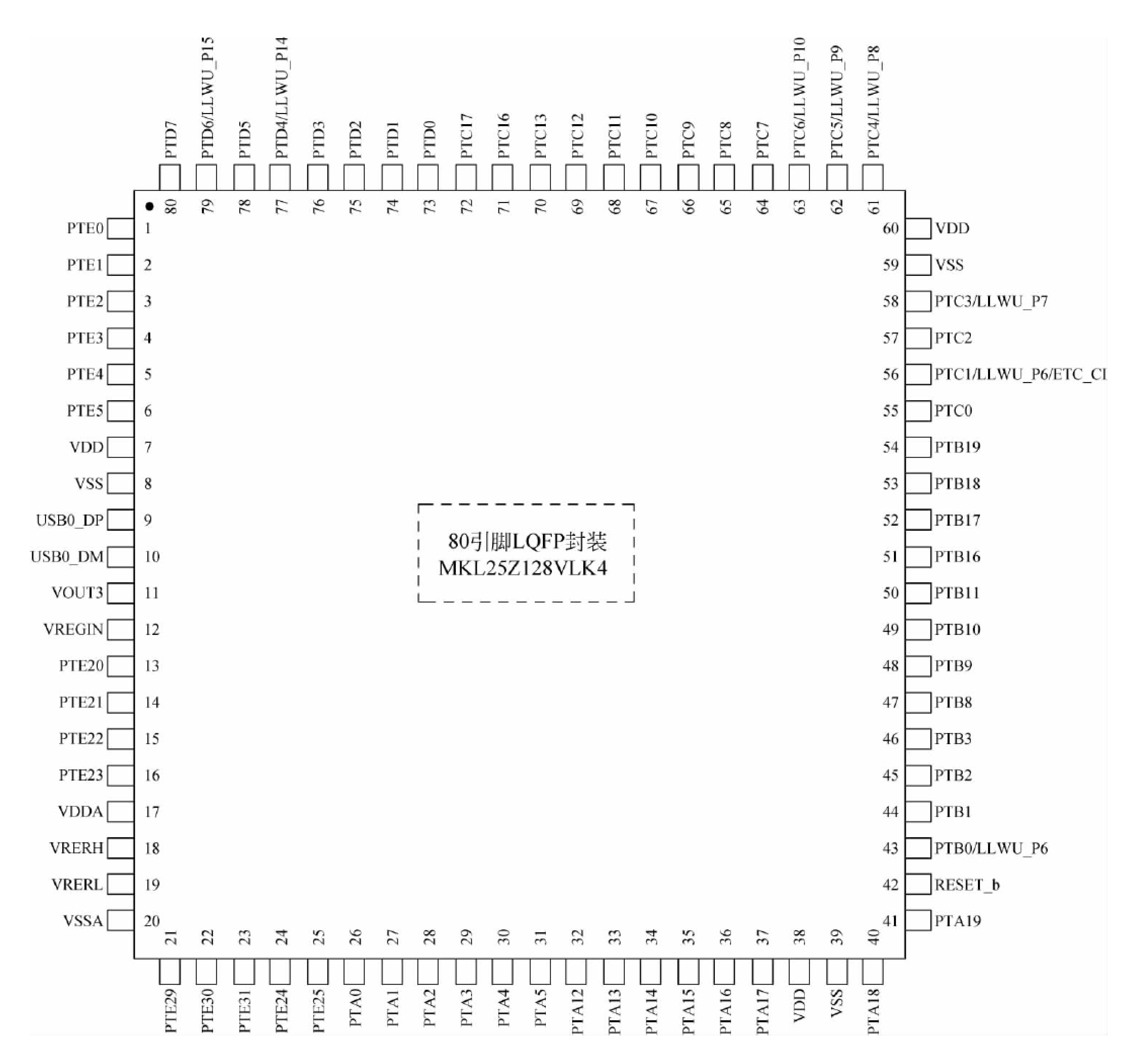


图 3-2　80 引脚 LQFP 封装 MKL25Z128VLK4 引脚图

脚，LQFP 封装 12 个。芯片使用多组电源引脚分别为内部电压调节器、I/O 引脚驱动、AD 转换电路等电路供电，内部电压调节器为内核和振荡器等供电。为了提供稳定的电源，MCU 内部包含多组电源电路，同时给出多处电源引出脚，便于外接滤波电容。为了电源平衡，MCU 提供了内部有共同接地点的多处电源引脚，供电路设计使用。

表 3-7　KL25/26 硬件最小系统引脚表

分类	引脚名	引脚号		功能描述
		KL25	KL26	
电源输入	VDD	7、38、60	3、30、48	电源，典型值：3.3V
	VSS	8、39、59	4、31、47	地，典型值：0V
	VDDA，VSSA	17、20	13、16	AD 模块的输入电源，典型值：（3.3V、0V）
	VREFH，VREFL	18、19	14、15	AD 模块的参考电压，典型值：（3.3V、0V）
	VREGIN	12	8	USB 模块的参考电压，典型值：5V
	VOUT33	11	7	USB 模块电源稳压器输出的电压，典型值：3.3V

续表

分类	引脚名	引脚号		功能描述
		KL25	KL26	
复位	RESET	42	34	双向引脚。有内部上拉电阻。作为输入,拉低可使芯片复位①
晶振	EXTAL,XTAL	40,41	32、33	分别为无源晶振输入、输出引脚
SWD 接口	SWD_CLK	26	22	SWD 时钟信号线
	SWD_DIO	29	25	SWD 数据信号线
引脚个数统计				硬件最小系统引脚均为17个

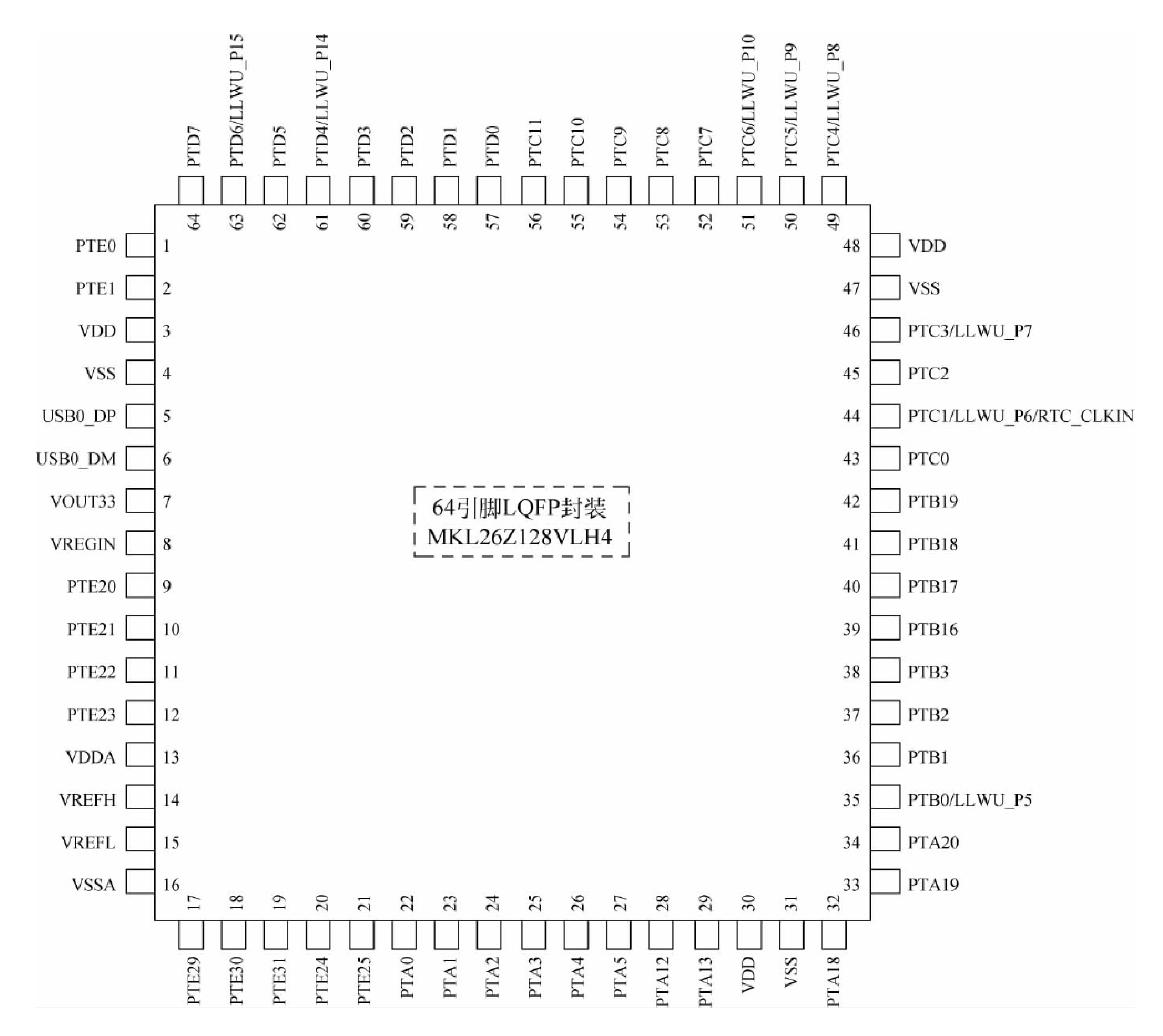


图 3-3　64 引脚 LQFP 封装 MKL26Z128VLH4 引脚图

① 拉低脉冲宽度需维持1.5个总线时钟周期以上,方能完成复位。作为输出,复位开始后,芯片内部电路驱动该引脚至少维持34个总线时钟周期的低电平。上电复位后,该引脚默认为RESET功能,复位完成后,可通过系统选项寄存器SIM_SOPT0的RSTPE位配置为其他功能,一般不建议这样做,最好就做复位引脚。

3.4.2 对外提供服务的引脚

除了需要我们为芯片服务的引脚(最小硬件系统引脚)之外,芯片的其他引脚为我们提供服务,也可称之为 I/O 端口资源类引脚,见表 3-8。这些引脚一般具有多种复用功能,**附录 A 给出了 KL25、KL26 芯片引脚功能复用表。**实际硬件设计时,必须依据该表,仔细斟酌引脚功能的使用,软件编程时,依据所使用的功能设定复用功能中的一种。因此,**读者需重点掌握该表的应用方法。**

表 3-8 KL25/26 对外提供 I/O 端口资源类引脚表

	KL25		KL26	
端口名	引脚数(63)	引脚名	引脚数(49)	引脚名
A	10	PTA[1～2、4～5],PTA[12～17]	8	PTA[0～5],PTA12,PTA13
B	12	PTB[0～3],PTB[8～11],PTB[16～19]	8	PTB[0～3],PTB[16～19]
C	16	PTC[0～13],PTC[16～17]	12	PTC[0～11]
D	8	PTD[0～7]	8	PTD[0～7]
E	15	PTE[0～5],PTE[20～25],PTE[29～31]	11	PTE[0～1],PTE[20～25],PTE[29～31]
其他	2	USB0_DM,USB0_DP	2	USB0_DM,USB0_DP
说明	这里统计 I/O 引脚不包括已被最小系统使用的引脚,但包含两个 SWD 的引脚。I/O 端口引脚最大输入电压为 0.7×VDD;最大输出电压 VDD,最大输出总电流 100mA。具体技术指标参见《KL26 数据手册》			

KL25(80 引脚 LQFP 封装)具有 63 个 I/O 引脚(包含两个 SWD 的引脚),KL26(64 引脚 LQFP 封装)具有 49 个 I/O 引脚(包含两个 SWD 的引脚)①。这些引脚均具有多个功能,在复位后,会立即被配置为高阻状态,且为通用输入引脚,有内部上拉功能。

随后各章以 KL25 为主进行讲解,但其内容完全适用于 KL26,网上教学资源中给出了 KL26 的程序。选用 KL26 的读者可使用这部分程序进行实验与实践。

3.5 KL25/26 硬件最小系统原理图

MCU 的硬件最小系统是指包括电源、晶振、复位、写入调试器接口等可使内部程序得以运行的、规范的、可复用的核心构件系统。使用一个芯片,必须完全理解其硬件最小系统。当 MCU 工作不正常时,首先就要查找最小系统中可能出错的元件。一般情况下,MCU 的硬件最小系统由电源、晶振及复位等电路组成。芯片要能工作,必须有电源与工作时钟;至于复位电路则提供不掉电情况下 MCU 重新启动的手段。随着 Flash 存储器制造技术的发展,大部分芯片提供了在板或在线系统(On System)的写入程序功能,即把空白芯片焊接到

① 写入器 SWD 使用的两个引脚在硬件最小系统表中与对外提供 I/O 端口资源类引脚表中重复列出,是因为这两个引脚在运行过程中作为其他功能使用是合适的。

电路板上后,再通过写入器把程序下载到芯片中。这样,硬件最小系统应该把写入器的接口电路也包含在其中。基于这个思路,KL25/26 芯片的硬件最小系统包括电源电路、复位电路、与写入器相连的 SWD 接口电路及可选晶振电路。**附录 B 给出了 KL25/26 硬件最小系统原理图。读者需彻底理解该原理图的基本内涵。**

3.5.1 电源及其滤波电路

电路中需要大量的电源类引脚用来提供足够的电流容量同时保持芯片电流平衡,所有的电源引脚必须外接适当的滤波电容抑制高频噪声。

电源(VDDx)与地(VSSx)包括很多引脚,如 VDDA、VSSA、VDD、VSS、VREFH 和 VREFL 等。至于外接电容,是由于集成电路制造技术所限,无法在 IC 内部通过光刻的方法制造这些电容。去耦是指对电源采取进一步的滤波措施,去除两级间信号通过电源互相干扰的影响,电源滤波电路可改善系统的电磁兼容性,降低电源波动对系统的影响,增强电路工作的稳定性。为标识系统通电与否,可以增加一个电源指示灯。

需要强调的是,虽然硬件最小系统原理图(附录 B)中的许多滤波电容被画在了一起,但实际布板时,需要各自接到靠近芯片的电源与地之间,才能起到良好的效果。

3.5.2 复位电路及复位功能

复位,意味着 MCU 一切重新开始。复位引脚为 RESET。若复位引脚有效(低电平),则会引起 MCU 复位。复位电路原理如下:正常工作时,复位引脚 RESET 通过一个 10kΩ 的电阻接到电源正极,所以应为高电平。若按下复位按钮,则 RESET 脚接地为低电平,导致芯片复位。若是系统重新上电,芯片内部电路会使 RESET 脚拉低,使芯片复位。KL25/26 的复位引脚是双向引脚,作为输入引脚,拉低可使芯片复位,作为输出引脚,上电复位期间有低脉冲输出,表示芯片已经复位完成。

从引起 MCU 复位的内部与外部因素来区分,复位可分为外部复位和内部复位两种。外部复位有上电复位、按下复位按钮复位。内部复位有看门狗定时器复位、低电压复位、软件复位等(见 13.6 节)。

从复位时芯片是否处于上电状态来区分,复位可分为冷复位和热复位。芯片从无电状态到上电状态的复位属于冷复位,芯片处于带电状态时的复位叫热复位。冷复位后,MCU 内部 RAM 的内容是随机的。而热复位后,MCU 内部 RAM 的内容会保持复位前的内容,即热复位并不会引起 RAM 中内容的丢失。

从 CPU 响应快慢来区分,复位还可分为异步复位与同步复位。异步复位源的复位请求一般表示一种紧要的事件,因此复位控制逻辑不等到当前总线周期结束,复位立即有效。异步复位源有上电、低电压复位等。同步复位的处理方法与异步复位不同:当一个同步复位源给出复位请求时,复位控制器并不使之立即起作用,而是等到当前总线周期结束之后,这是为了保护数据的完整性。在该总线周期结束后的下一个系统时钟的上升沿时,复位才有效。同步复位源有看门狗定时器、软件等。

3.5.3 晶振电路

KL25/26 芯片可使用内部晶振或外部晶振两种方式为 MCU 提供工作时钟。

KL25/26芯片含有内部时钟源(IRC),频率分为慢速32.768kHz和快速4MHz,慢速内部时钟误差在0.6%以内,而快速内部时钟误差在3%以内。通过编程,最大可产生48MHz内核时钟及24MHz总线时钟。使用内部时钟源可略去外部晶振电路。

若时钟源需要更高的精度,可自行选用外部晶振,例如图3-4给出外接8MHz无源晶振的晶振电路接法,晶振连接在晶振输入引脚EXTAL0、晶振输出引脚XTAL0之间。有关配置及具体编程见第13.1节(时钟系统)。

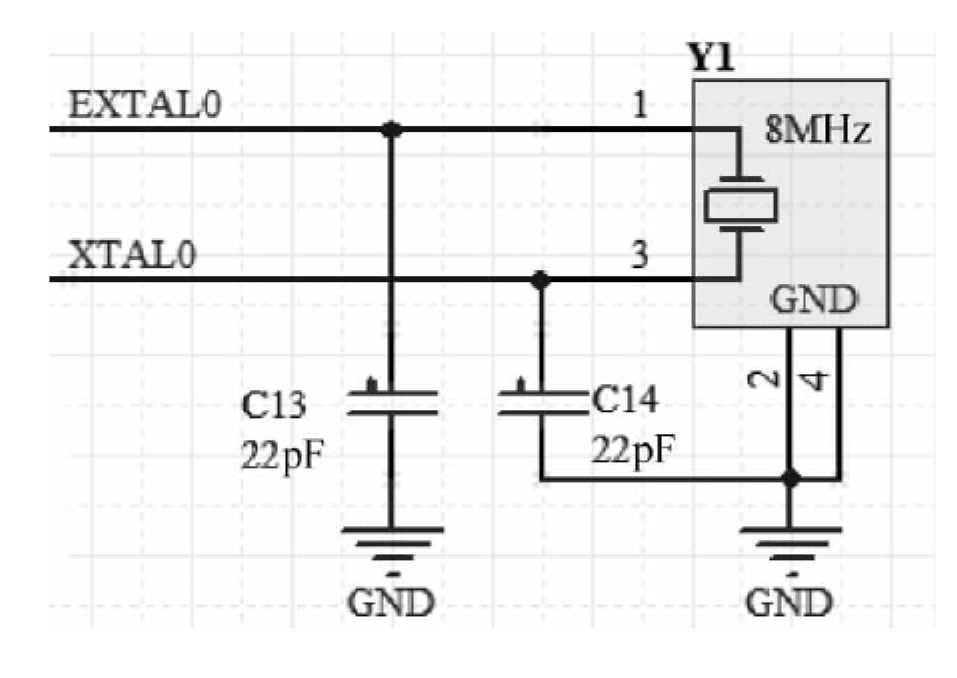


图3-4　晶振电路

3.5.4　SWD接口电路

KL25/26芯片的调试接口SWD是基于CoreSight架构,该架构在限制输出引脚和其他可用资源情况下,提供了最大的灵活性。CoreSight是ARM定义的一个开放体系结构,以使SOC设计人员能够将其他IP内核的调试和跟踪功能添加到CoreSight基础结构中。通过SWD接口可以实现程序下载和调试功能。SWD接口只需两根线,数据输入/输出线(DIO)和时钟线(CLK)。附录B最小硬件系统原理图中,给出了SWD调试接口电路,连接到KL25/26芯片的SWD_DIO与SWD_CLK两个引脚。可根据实际需要增加地、电源以及复位信号线。

小　　结

本章主要给出了KL25/KL26存储映像、中断源、引脚图及引脚表,重点是给出了硬件最小系统,完成了MCU的基础硬件入门。

(1) 基于ARM Cortex-M内核的Kinetis系列微控制器,主要有K、L、M、W、E、EA及V系列,这些系列的特点各不相同,适用于不同应用领域。

(2) KL系列的一个具体MCU型号标识含有质量状态、系列号、内核类型、内部Flash大小、温度范围、封装类型、CPU最高频率、包装类型等信息。

(3) 关于KL25/26系列的存储映像与中断源。其片内Flash大小为128KB,地址范围:0x0000_0000 ～ 0x0001_FFFF,用来存放中断向量、程序代码、常数等;片内RAM大小16KB,地址范围:0x1FFF_F000～0x2000_2FFF,用来存储全局变量、临时变量(堆栈空间)等;KL25/26最多支持48个中断源,为中断向量表中提供物理基础,由于中断的内容会在后面章节详细介绍,本章了解即可。

(4) 关于硬件最小系统。一个芯片的硬件最小系统是指可以使内部程序运行所必需的最低规模的外围电路,也可以包括写入器接口电路。使用一个芯片,必须完全理解其硬件最小系统。硬件最小系统引脚是我们必须为芯片提供服务的引脚,包括电源、晶振、复位、SWD接口。读者需充分理解附录B的硬件最小系统原理图。该图可从5个部分来理解,第一,首先需要为芯片提供电源,直流3.3V,所有的电源引出脚与地之间应在靠近芯片的地方

接滤波电容(去耦电容),因为电容有通交流阻直流的特性,因此用来抑制高频噪声,使供电更加稳定;第二,需要给芯片提供晶振,芯片工作需要一个由晶振提供的时钟信号;第三,复位引脚要加上拉电阻,平时电平拉高,需要复位时与地导通使电平拉低,让芯片复位,从而使芯片复位;第四,是SWD写入器接口,为了将程序写入芯片,需要写入器接口引脚;第五,其他引脚引出虚线之外,就为我们提供服务了。

(5)学习第5章之后,再回头来理解为什么这样画原理图。我们的目标是,所有使用该芯片的应用系统,硬件最小系统原理图可复用,第5章称之为“核心构件”。

习　　题

1. 简述ARM Cortex-M0＋ KL系列MCU的型号标识。

2. 给出所学芯片的RAM、Flash的地址范围,说明堆栈空间、全局变量、常量、程序分别存放于RAM中还是Flash中。芯片初始化时,SP值应为何值,说明原因。

3. 简要阐述硬件电路中滤波电路、耦合电路的具体作用。

4. 解释最小硬件系统概念,并结合所学芯片的开发板,归纳实现最小系统需要的引脚资源。

5. 所学芯片的开发板中使用什么标准调试接口?具体如何实现?

6. 所学芯片的开发板中具有哪些功能接口?如何进行测试?

7. 概要给出所学芯片的最小系统原理图的各部分基本原理。

8. 自行找一个型号MCU,给出设计硬件最小系统的基本步骤,并参考本章样例画出原理图。

第4章　GPIO及程序框架

本章导读：本章是全书的重点和难点之一，需要花工夫透彻理解，达到快速且规范入门的目的。主要内容有：①给出通用I/O基本概念及连接方法；②简明扼要地介绍了KL的端口控制模块与GPIO模块的编程结构，举例通过给映像寄存器赋值的方法，点亮一盏小灯的编程步骤，以便理解底层驱动的含义与编程方法；③阐述设计底层驱动构件的必要性及基本方法，给出GPIO驱动构件设计方法，这是第一个基础构件设计样例；④给出利用GPIO驱动构件设计Light应用构件的方法，这是第一个利用基础驱动构件设计应用构件的样例；⑤给出第一个构件化编程框架、文件组织、上电启动执行过程分析；⑥给出一个规范的汇编工程样例，供汇编入门使用。网上教学资源资料中给出了最小系统硬件资料、开发环境及工程调试方法介绍。

本章参考资料：4.2.1节(端口控制模块)总结自《KL参考手册》的第11章，引脚驱动能力来自《KL数据手册》5.2.3节；4.2.2节(GPIO模块)总结自《KL参考手册》的第41章。

4.1　通用I/O接口基本概念及连接方法

下面利用GPIO编程作为第一个程序入门样例，并以此为基础给出工程框架，阐述基本编程规范。

1. I/O接口的概念

I/O接口，即输入/输出(Input/Output)接口，是MCU同外界进行交互的重要通道，MCU与外部设备的数据交换通过I/O接口来实现。I/O接口是一个电子电路，其内由若干专用寄存器和相应的控制逻辑电路构成。接口的英文单词是Interface，另一个英文单词是Port。但有时把interface翻译成“接口”，而把port翻译成“端口”。从中文字面看，接口与端口似乎有点儿区别，但在嵌入式系统中它们的含义是相同的。有时把I/O引脚称为接口(Interface)，而把用于对I/O引脚进行编程的寄存器称为端口(Port)，实际上它们是紧密相连的。因此，不必深究它们之间的区别。有些书中甚至直接称I/O接口(端口)为I/O口。在嵌入式系统中，接口千变万化，种类繁多，有显而易见的人机交互接口，如操纵杆、键盘、显示器；也有无人介入的接口，如网络接口、机器设备接口等。

2. 通用I/O

所谓通用I/O，也记为GPIO(General Purpose I/O)，即基本的输入/输出，有时也称并行I/O，或普通I/O，它是I/O的最基本形式。本书中使用正逻辑，电源(Vcc)代表高电平，对应数字信号“1”；地(GND)代表低电平，对应数字信号“0”。作为通用输入引脚，MCU内部程序可以通过端口寄存器**获取该引脚状态**，以确定该引脚是“1”(高电平)或“0”(低电平)，即开关量输入。作为通用输出引脚，MCU内部程序通过端口寄存器**控制该引脚状态**，使得

引脚输出“1”(高电平)或“0”(低电平),即开关量输出。大多数通用I/O引脚可以通过编程来设定其工作方式为输入或输出,称为双向通用I/O。

3. 上拉下拉电阻与输入引脚的基本接法

芯片输入引脚的外部有三种不同的连接方式:带上拉电阻的连接、带下拉电阻的连接和“悬空”连接。通俗地说,若MCU的某个引脚通过一个电阻接到电源(V_{CC})上,这个电阻被称为“上拉电阻”;与之相对应,若MCU的某个引脚通过一个电阻接到地(GND)上,则相应的电阻被称为“下拉电阻”。这种做法使得悬空的芯片引脚被上拉电阻或下拉电阻初始化为高电平或低电平。根据实际情况,上拉电阻与下拉电阻可以取值在1～10kΩ之间,其阻值大小与静态电流及系统功耗有关。

图4-1给出了一个MCU的输入引脚的三种外部连接方式,假设MCU内部没有上拉或下拉电阻,图中的引脚I3上的开关K3采用悬空方式连接就不合适,因为K3断开时,引脚I3的电平不确定。在图4-1中,R1 >> R2,R3 << R4,各电阻的典型取值为:R1=20kΩ,R2=1kΩ,R3=10kΩ,R4=200kΩ。

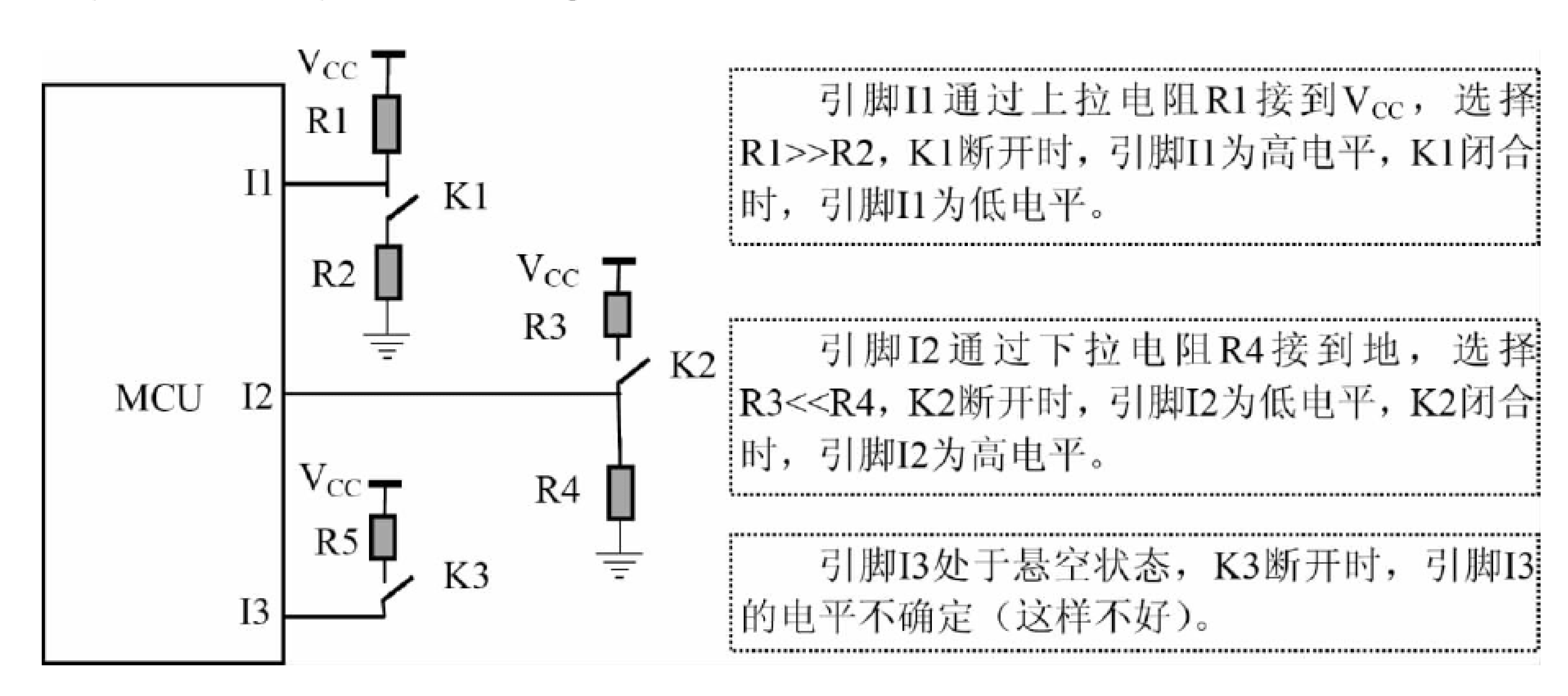


图4-1　通用I/O引脚输入电路接法举例

4. 输出引脚的基本接法

作为通用输出引脚,MCU内部程序向该引脚输出高电平或低电平来驱动器件工作,即开关量输出,如图4-2所示,输出引脚O1和O2采用了不同的方式驱动外部器件。一种接法是O1直接驱动发光二极管LED,当O1引脚输出高电平时,LED不亮;当O1引脚输出低电平时,LED点亮。这种接法的驱动电流一般在2～10mA。另一种接法是O2通过一个NPN三极管驱动蜂鸣器,当O2引脚输出高电平时,三极管导通,蜂鸣器响;当O2引脚输出低电平时,三极管截止,蜂鸣器不响。这种接法可以用O2引脚上的几个毫安的控制电流驱动高达100mA的驱动电流。若负载需要更大的驱动电流,就必须采用光电隔离外加其他驱动电路,但对MCU编程来说,没有任何影响。

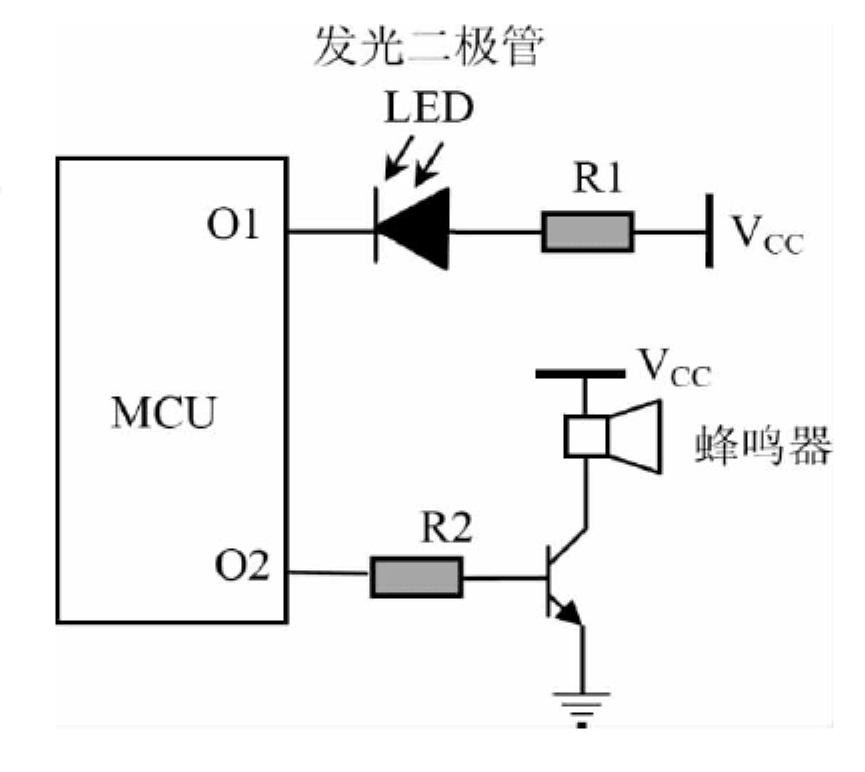


图4-2　通用I/O引脚输出电路

4.2　端口控制模块与 GPIO 模块的编程结构

为了实现快速入门，本节将利用 MCU 的一个引脚控制一盏如图 4-2 所示的发光二极管 LED。为此，需要掌握配置引脚具体功能的端口控制模块（PORT）以及可控制引脚高低电平输出的 GPIO 模块的基本用法。

4.2.1　端口控制模块——决定引脚复用功能

KL25 的大部分引脚具有复用功能，可以通过端口控制模块（Port Control and Interrupts，PORT）提供的寄存器编程指定其为某一具体功能。

PORT 模块内含三类寄存器，分别是引脚控制寄存器（Pin Control Register）、全局引脚控制寄存器（Global Pin Control Register）、中断状态标志寄存器（Interrupt Status Flag Register）。

1. 寄存器映像地址分析

KL25 芯片有 5 个端口 A～E。每个端口有 32 个引脚控制寄存器 PORTx_PCRn（其中 x＝A～E，n＝0～31），两个全局引脚控制寄存器（PORTx_GPCLR、PORTx_GPCHR）、一个中断状态标志寄存器（PORTx_ISFR）。以下地址分析计算均为十六进制，为书写简化起见，在不致引起歧义的情况下，略去十六进制前缀“0x”不写。

每个端口有 32 个引脚控制寄存器 PORTx_PCRn。端口 x 的基地址＝4004_9000＋x×1000（x＝A～E，对应 0～4）。端口 x 的每个引脚控制寄存器 PORTx_PCRn 的地址为＝4004_9000＋x×1000＋n×4（x＝A～E，对应 0～4，n＝0～31）。这样 5 个端口，共 5×32＝160 个引脚控制寄存器，每个引脚控制寄存器的地址很容易计算出来。例如，PORTA_PCR1 的地址为：4004_9000＋0×1000＋1×4＝4004_9004。

每个端口有两个全局引脚控制寄存器。全局引脚控制寄存器（低）PORTx_GPCLR，地址＝4004_9080＋x×1000（x＝A～E，对应 0～4）；全局引脚控制寄存器（高）PORTx_GPCHR，地址＝4004_9084＋x×1000（x＝A～E，对应 0～4）。

每个端口有一个中断状态标志寄存器。地址＝4004_90A0＋x×1000（x＝A～E，对应 0～4）。

2. 相关名词解释

（1）模拟引脚（Analog Pin）是指不能够配置成 GPIO 的引脚，如 RESET、EXTAL 及 XTAL 等引脚。KL25 的所有模拟引脚在芯片内部都有 ESD（Electro-Static Discharge，静电阻抗器）保护二极管连接到 Vss 和 VDD。

（2）数字引脚（Digital Pin）是指能够被配置成 GPIO 的引脚。所有的数字引脚都会通过一个 ESD 保护二极管连接到 Vss。

（3）无源滤波器（Passive Filter）是由电容器、电抗器和电阻器适当组合而成，并兼有无功补偿和调压功能的滤波器。可滤除一次或多次谐波，最简单的无源滤波器结构是将电感与电容串联。

（4）引脚驱动能力（Drive Strength）是指引脚放出或吸入电流的承受能力，一般用 mA 单位度量。

(5) 转换速率(Slew Rate)是指电压在高低电平间转换的时间间隔，一般用 ns 单位度量。

(6) 数字输出(Digital Output)是指芯片引脚只能输出高电平(逻辑 1)和低电平(逻辑 0)两个电压值。

(7) 数字输入(Digital Input)是指芯片引脚只能接收并识别高电平(逻辑 1)和低电平(逻辑 0)两个电压值。

(8) 引脚复用槽(Pin Multiplexing Slot)是指信号复用装置与引脚之间的接口，引脚通过连接不同的信号复用槽可以配置成不同的功能。

(9) w1c 是指对某位写 1 而使得该位清 0，俗称写 1 清 0。

3. 引脚控制寄存器

重点掌握本寄存器的使用，重中之重掌握本寄存器 D10～D8(MUX)——引脚复用控制字段，它决定引脚复用何种功能。

每个端口的每个引脚均有一个对应的引脚控制寄存器(PORTx_PCRn)，可以配置引脚中断或 DMA 传输请求，可以配置引脚为 GPIO 功能或其他功能，可以配置是否启用上拉或下拉，可以配置选择输出引脚的驱动强度，可以配置选择输入引脚是否使用内部滤波等。

<table>
<tr><th>数据位</th><th>D31</th><th>D30</th><th>D29</th><th>D28</th><th>D27</th><th>D26</th><th>D25</th><th>D24</th><th>D23</th><th>D22</th><th>D21</th><th>D20</th><th>D19</th><th>D18</th><th>D17</th><th>D16</th></tr>
<tr><td>读</td><td colspan="7">0</td><td>ISF</td><td colspan="4">0</td><td colspan="4" rowspan="2">IRQC</td></tr>
<tr><td>写</td><td colspan="7">—</td><td>w1c</td><td colspan="4">—</td></tr>
<tr><td>复位</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><th>数据位</th><th>D15</th><th>D14</th><th>D13</th><th>D12</th><th>D11</th><th>D10</th><th>D9</th><th>D8</th><th>D7</th><th>D6</th><th>D5</th><th>D4</th><th>D3</th><th>D2</th><th>D1</th><th>D0</th></tr>
<tr><td>读</td><td colspan="5">0</td><td colspan="3" rowspan="2">MUX</td><td>0</td><td rowspan="2">DSE</td><td>0</td><td rowspan="2">PFE</td><td>0</td><td rowspan="2">SRE</td><td rowspan="2">PE</td><td rowspan="2">PS</td></tr>
<tr><td>写</td><td colspan="5">—</td><td>—</td><td>—</td><td>—</td></tr>
<tr><td>复位</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>X</td><td>X</td><td>X</td><td>0</td><td>X</td><td>0</td><td>X</td><td>0</td><td>X</td><td>X</td><td>X</td></tr>
</table>

其中，"X"表示复位后状态不确定。下面给出有关功能说明，未说明的位或字段均为保留(只读，值为 0)。

D24(ISF)——中断状态标志(只读)。数字引脚模式下有效。ISF＝0，未检测到引脚中断；ISF＝1，检测到引脚中断。向该位写 1，可清除中断状态标志。若引脚配置为 DMA 请求方式，在完成 DMA 请求传输后，将自动清除中断状态标志。如果引脚被配置为电平触发的中断，引起中断的电平若一直有效，该标志将一直保持置位，即使被清除后也会立即置位。

D19～D16(IRQC)——中断配置情况(读/写)。数字引脚模式下有效。IRQC＝0000，关闭引脚中断/DMA 请求；IRQC＝0001～0011——分别对应上升沿、下降沿、沿跳变，触发 DMA 请求；0100——保留；1000～1100——分别对应逻辑低电平(逻辑 0)、上升沿、下降沿、沿跳变、高电平(逻辑 1)，触发引脚中断。其他值——保留。**特别注意：并不是所有 KL25 的引脚均可配置为中断功能，只有 A 口、D 口的引脚具有上述这种中断功能。**

D10～D8(MUX)——**引脚复用控制(读/写)**。不是所有引脚都支持引脚复用槽。MUX＝000，引脚不配置(模拟引脚)；MUX＝001，配置引脚为通用输入输出(GPIO)功能；MUX＝010～111，分别配置引脚的功能为第 2 到第 7 功能(具体功能见芯片参考手册 10.3

节)。**附录A给出了KL25/26引脚复用功能情况。**

D6(DSE)——驱动能力使能位(读/写)。表明引脚被配置为数字输出时的驱动能力状况,数字引脚模式下有效。DSE=0,低驱动能力;DSE=1,高驱动能力。由数据手册可知,KL25低驱动能力是5mA,高驱动能力是18mA,但并不是所用引脚均可配置成高驱动能力,实际使用时,需查数据手册。

D4(PFE)——无源滤波使能位(读/写)。数字引脚模式下有效。PFE=0,相应的引脚禁止无源输入滤波;PFE=1,相应的引脚启用无源输入滤波。具体滤波性能需参考数据手册,KL25数据手册未给出无源滤波性能,可以不启用此功能,必要时自行外接滤波电路。

D2(SRE)——转换速率使能位(读/写)。数字引脚模式下有效。0——引脚配置成快转换速率;1——引脚配置成慢转换速率。查数据手册,KL25的转换速率最慢为16ns,此项设定未给出具体表述,一般使用默认值0。

D1(PE)——上拉或下拉使能位(读/写)。数字引脚模式下有效。0——相应的引脚关闭内部上拉或下拉电阻;1——相应的引脚启用内部上拉或下拉电阻,引脚作为数字输入。

D0(PS)——上拉或下拉选择(读/写)。数字引脚模式下有效。PS=0,如果PE=1,引脚下拉电阻使能;PS=1,如果PE=1,引脚上拉电阻使能。KL25内部上下拉电阻大小为20~50kΩ。

4. 全局引脚控制寄存器

本寄存器只需了解即可。

每个端口的全局引脚控制寄存器有两个,分别为PORTx_GPCLR、PORTx_GPCHR,为只写寄存器,读出总为0。每个寄存器的高16位被称为全局引脚写使能字段(Global Pin Write Enable,GPWE),低16位被称为全局引脚写数据字段(Global Pin Write Data,GPWD)。如果设定GPWE=0xFFFF,则GPWD字段的16位就被写入到一整组引脚控制寄存器的低16位中。KL25芯片每个端口有32个引脚控制寄存器,分为两组:低引脚控制寄存器组(15~0)和高引脚控制寄存器组(31~16),全局引脚控制寄存器PORTx_GPCLR配置低引脚控制寄存器组(15~0),而全局引脚控制寄存器PORTx_GPCHR配置高引脚控制寄存器组(31~16)。这样可以实现一次配置16个功能相同的引脚,提高了编程效率。GPWE字段中的16位对应16个引脚控制寄存器,如果GPWE字段的部分位为0,则引脚控制寄存器组中对应的引脚控制寄存器不被配置。全局引脚控制寄存器不能配置引脚控制寄存器的高16位,因此,不能使用该功能配置引脚中断。

5. 中断状态标志寄存器

本寄存器要求基本理解。

数字引脚模式下,每个引脚的中断模式可以独立配置,在引脚控制寄存器IRQC字段可配置选择:中断禁止(复位后默认);高电平、低电平、上升沿、下降沿、沿跳变触发中断;上升沿、下降沿、沿跳变触发DMA请求。支持低功耗模式下唤醒。

每个端口的中断状态标志寄存器(PORTx_ISFR),对应该口的32个引脚,相应位为1,表明配置的中断已经被检测到,反之没有。各位具有写1清0特性。

4.2.2 GPIO模块——对外引脚与内部寄存器

1. KL25的GPIO引脚

KL25的大部分引脚具有多重复用功能,可以通过4.2.1节给出的寄存器编程来设定

使用其中某一种功能。本节给出作为GPIO功能时的编程结构。80引脚封装的KL25芯片的GPIO引脚分为5个端口,标记为A、B、C、D、E,共含61个引脚。端口作为GPIO引脚时,逻辑1对应高电平,逻辑0对应低电平。GPIO模块使用系统时钟,从实时性细节来说,当作为通用输出时,高/低电平出现在时钟上升沿。每个口实际可用的引脚数因封装不同而有差异,下面给出各口可作为GPIO功能的引脚数目及引脚名称。

(1) A口有10个引脚,分别记为PTA1、PTA2、PTA4~5、PTA12~17;

(2) B口有12个引脚,分别记为PTB0~3、PTB8~11、PTB16~19;

(3) C口有16个引脚,分别记为PTC0~13、PTC16~17;

(4) D口有8个引脚,分别记为PTD0~7;

(5) E口有15个引脚,分别记为PTE0~5、PTE20~25、PTE29~31。

处理器使用零等待方式,以最高性能访问通用输入输出。GPIO寄存器支持8位、16位及32位接口。在运行、等待、调试模式下,GPIO工作正常,在停止模式下,GPIO停止工作。

2. GPIO寄存器

每个GPIO口均有6个寄存器,5个GPIO口共有30个寄存器。A、B、C、D、E各口寄存器的基地址分别为400F_F000h、400F_F040h、400F_F0080h、400F_F0C0h、400F_F100h,所以各口基地址相差40h。各GPIO口的6个寄存器分别是数据输出寄存器、输出置1寄存器、输出清0寄存器、输出反转寄存器、数据输入寄存器、数据方向寄存器。其中,输出寄存器的地址就是口的基地址,其他寄存器的地址依次加4。所有寄存器均为32位宽度,复位时均为0000_0000h。表4-1给出了A口的6个寄存器的基地址、偏移地址、绝对地址、寄存器名、访问特性、功能描述。其他各口功能与编程方式完全一致,只是相应寄存器名与寄存器地址不同,其中,寄存器名只要把其中的A口的字母“A”分别改为B、C、D、E即可获得,地址按上述给出的规律计算。

表4-1　PORTA寄存器

基地址	地址偏移		绝对地址	寄存器名	访问	功能描述
	字	字节				
400F_F000h	0	0h	400F_F000h	数据输出寄存器(GPIOA_PDOR)	R/W	当引脚被配置为输出时,若某一位为0,则对应引脚输出低电平;为1,则对应引脚输出高电平
	1	4h	400F_F004h	输出置1寄存器(GPIOA_PSOR)	W	写0不改变输出寄存器相应位,写1将输出寄存器相应位置1
	2	8h	400F_F008h	输出清0寄存器(GPIOA_PCOR)	W	写0不改变输出寄存器相应位,写1将输出寄存器相应位清0
	3	Ch	400F_F00Ch	输出取反寄存器(GPIOA_PTOR)	W	写0不改变输出寄存器相应位,写1将输出寄存器的相应位取反(即1变0,0变1)
	4	10h	400F_F010h	数据输入寄存器(GPIOA_PDIR)	R	若读出为0,表明相应引脚上为低电平;若读出为1,表明相应引脚上为高电平
	5	14h	400F_F014h	数据方向寄存器(GPIOA_PDDR)	R/W	各位值决定了相对应的引脚为输入还是输出。若其某位设定为0,则相对应的引脚为输入;为1,则相对应的引脚为输出

4.2.3　GPIO 基本编程步骤与基本打通程序

1. GPIO 基本编程步骤

要使芯片某一引脚为 GPIO 功能，并定义为输入/输出，随后进行应用，基本编程步骤如下。

(1) 通过端口控制模块(PORT)的引脚控制寄存器 PORTx_PCRn 的引脚复用控制字段(MUX)设定其为 GPIO 功能(即令 MUX＝0b001)。

(2) 通过 GPIO 模块相应口的“数据方向寄存器”来指定相应引脚为输入或输出功能。若指定位为 0，则为对应引脚输入；若指定位为 1，则为对应引脚输出。

(3) 若是输出引脚，则通过设置“数据输出寄存器”来指定相应引脚输出低电平或高电平，对应值为 0 或 1。也可通过“输出置位寄存器”“输出清位寄存器”“输出取反寄存器”改变引脚状态，参见表 4-1 中关于寄存器的说明。

(4) 若是输入引脚，则通过“数据输入寄存器”获得引脚的状态。若指定位为 0，表示当前该引脚上为低电平；若为 1，则为高电平。

2. 理解 GPIO 基本编程步骤举例——基本打通程序

举例说明：设 PORTB 口的 19 脚接一个发光二极管，低电平点亮(图 4-2 中的接法)。现在要点亮这个发光二极管，步骤如下。

1) 计算引脚控制寄存器 PCR、数据方向寄存器 PDDR、输出寄存器 PDOR 的地址

(1) 计算给出 PORTB19 引脚控制寄存器地址。从 4.2.1 节的端口控制模块可知，PORTB 端口的引脚控制寄存器基地址为 0x4004A000u，其中，后缀 u 表示无符号数，给出不优化的 32 位指针变量 portB_ptr：

```
volatile uint_32 *portB_ptr = (uint_32 *)0x4004A000u;
```

PORTB19 引脚控制寄存器地址＝基地址＋偏移量：

```
volatile uint_32 *portB_PCR_19 = portB_ptr + 19;
```

这里是 19，而不是 19×4，由于定义了 32 位指针，portB_ptr 加 1 相当于地址加 4。portB_ptr 加 19 代表了 portB_ptr 地址加上 19×4。

(2) 计算给出 PORTB 的数据方向寄存器、输出寄存器的地址。PORTB 端口(作为 GPIO 功能)的基地址为 0x400FF040u：

```
volatile uint_32 *gpioB_ptr = (uint_32 *)0x400FF040u;
```

参考表 4-1，PORTB 的数据方向寄存器地址＝基地址＋偏移量，PORTB 的数据输出寄存器地址＝基地址＋偏移量：

```
volatile uint_32 *portB_PDDR=gpioB_ptr+5;
volatile uint_32 *portB_PDO=gpioB_ptr+0;
```

2）设置 PORTB19 引脚为 GPIO 输出引脚并输出数据

(1) 令相应引脚控制寄存器的 10～8 位(MUX 位段)为 0b001，其他位保持：

```
* portB_PCR_19 &= 0b1111 1111 1111 1111 1111 1000 1111 1111;        //清 MUX 位段
* portB_PCR_19 |= 0b0000 0000 0000 0000 0000 0001 0000 0000;
```

(2) 通过令 PORTB 的方向寄存器相应位为 1，定义 PORTB19 引脚为输出：

```
* portB_PDDR |= (1<<19);
```

(3) 通过 PORTB 的输出寄存器相应位赋 0，使 PORTB19 引脚输出低电平：

```
* portB_PDO &= ~ (1<<19);
```

这样这个发光二极管就亮起来了。这种编程方法的样例，在本书网上教学资源的“..\ch04-Light\KL25-Light(Simple)\08_Source\main.c”文件中可以看到，用记事本就可打开该文件查看。安装开发环境，参考网上教学资源中< 01-Document >文件夹下关于软件工具使用方法的说明，利用开发环境，编译该工程，将机器码下载到硬件评估系统中，可以执行该程序。可以采用单步调试的方式，观察执行情况，以便理解实际映像寄存器与硬件是如何关联对应的，这样就理解了软件是如何控制硬件的。

不论如何，学到这里，应该进行实验。通过实验，理解基本原理，学会软件、硬件工具的使用与基本调试方法。

需要进一步说明的是，这样编程只是为了理解 GPIO 的基本编程方法，实际并不使用。芯片有那么多引脚，不可能这样编程，要把对底层硬件的操作用构件把它们封装起来，给出函数名与接口参数，供实际编程时使用。4.3 节将阐述底层驱动构件封装方法与基本规范。

4.3　GPIO 驱动构件封装方法与驱动构件封装规范

4.3.1　设计 GPIO 驱动构件的必要性及 GPIO 驱动构件封装要点分析

1. 设计 GPIO 驱动构件的必要性

软件构件(Software Component)技术的出现，为实现软件构件的工业化生产提供了理论与技术基石。将软件构件技术应用到嵌入式软件开发中，可以大大提高嵌入式开发的开发效率与稳定性。软件构件的封装性、可移植性与可复用性是软件构件的基本特性，采用构件技术设计软件，可以使软件具有更好地开放性、通用性和适应性。特别是对于底层硬件的驱动编程，只有封装成底层驱动构件，才能减少重复劳动，使广大 MCU 应用开发者专注于应用软件稳定性与功能设计上。因此，必须把底层硬件驱动设计好、封装好。

以 KL25 的 GPIO 为例，它有 61 个引脚可以作为 GPIO，分布在 5 个端口，不可能使用直接地址去操作相关寄存器，那样无法实现软件移植与复用。应该把对 GPIO 引脚的操作封装成构件，通过函数调用与传参的方式实现对引脚的干预与状态获取，这样的软件才便于

维护与移植，因此设计 GPIO 驱动构件十分必要。同时，底层驱动构件的封装，也为在操作系统下对底层硬件的操作提供了基础。

2. GPIO 驱动构件封装要点分析

同样以 GPIO 驱动构件为例，进行封装要点分析。即分析应该设计哪几个函数及入口参数。GPIO 引脚可以被定义成输入、输出两种情况：若是输入，程序需要获得引脚的状态(逻辑 1 或 0)；若是输出，程序可以设置引脚状态(逻辑 1 或 0)。MCU 的 PORT 模块分为许多端口，每个端口有若干引脚。GPIO 驱动构件可以实现对所有 GPIO 引脚统一编程。GPIO 驱动构件由 gpio.h、gpio.c 两个文件组成，如要使用 GPIO 驱动构件，只需要将这两个文件加入到所建工程中，由此方便了对 GPIO 的编程操作。

1) 模块初始化(gpio_init)

由于芯片引脚具有复用特性，应把引脚设置成 GPIO 功能；同时定义成输入或输出；若是输出，还要给出初始状态。所以 GPIO 模块初始化函数 gpio_init 的参数为哪个引脚、是输入还是输出、若是输出其状态是什么，函数不必有返回值。其中，引脚可用一个 16 位数据描述，高 8 位表示端口号，低 8 位表示端口内的引脚号。这样 GPIO 模块初始化函数原型可以设计为：

```
void gpio_init(uint_16 port_pin, uint_8 dir, uint_8 state)
```

其中，uint_8 是无符号 8 位整型的别名，uint_16 是无符号 16 位整型的别名，其定义在工程文件夹下的"..\07_Soft_Component\Common\common.h"文件中，本书后面不再特别说明。

2) 设置引脚状态(gpio_set)

对于输出，希望通过函数设置引脚是高电平(逻辑 1)还是低电平(逻辑 0)。入口参数应该是哪个引脚，输出其状态是什么，函数不必有返回值。这样设置引脚状态的函数原型可以设计为：

```
void gpio_set(uint_16 port_pin, uint_8 state)
```

3) 获得引脚状态(gpio_get)

对于输入，希望通过函数获得引脚的状态是高电平(逻辑 1)还是低电平(逻辑 0)，入口参数应该是哪个引脚，函数需要返回值引脚状态。这样设置引脚状态的函数原型可以设计为：

```
uint_8 gpio_get(uint_16 port_pin)
```

4) 引脚状态反转(void gpio_reverse)

类似的分析，可以设计引脚状态反转函数的原型为：

```
void gpio_reverse(uint_16 port_pin)
```

5) 引脚上下拉使能函数(void gpio_pull)

若引脚被设置成输入，还可以设定内部上下拉，KL25 内部上下拉电阻大小为 20～

50kΩ。引脚上下拉使能函数的原型为：

```
void gpio_pull(uint_16 port_pin, uint_8 pullselect)
```

这些函数基本满足了对 GPIO 操作的基本需求。还有中断使能与禁止[①]、引脚驱动能力等函数，这些是比较深的内容，可暂时略过，使用或深入学习时参考 GPIO 构件即可。要实现 GPIO 驱动构件的这几个函数，给出清晰的接口、良好的封装、简洁的说明与注释、规范的编程风格等，需要一些准备工作，4.3.2 节将给出构件封装基本规范与前期准备。

4.3.2 底层驱动构件封装规范概要与构件封装的前期准备

底层驱动构件封装规范见 5.3 节，本节给出概要与前期准备，以便读者在认识第一个构件前以及在开始设计构件时，少走弯路，做出来的构件符合基本规范，便于移植、复用、交流。

1. *底层驱动构件封装规范概要*

1）底层驱动构件的组成、存放位置与内容

每个构件由头文件(.h)与源文件(.c)两个独立文件组成，放在以构件名命名的文件夹中。驱动构件头文件(.h)中仅包含对外接口函数的声明，是构件的使用指南。以构件名命名。例如，GIPO 构件命名为 gpio(使用小写，目的是与内部函数名前缀统一)。设计好的 GPIO 构件存放于“..KL25 共用驱动\KL25 底层驱动构件\gpio”文件夹中，供复制使用。基本要求是调用者只看头文件即可使用构件。对外接口函数及内部函数的实现在构件源程序文件(.c)中。同时应注意，头文件声明对外接口函数的顺序与源程序文件实现对外接口函数的顺序应保持一致。源程序文件中内部函数的声明，放在外接口函数代码的前面，内部函数的实现放在全部外接口函数代码的后面，以便提高可阅读性与可维护性。

一个具体的工程中，在本书给出的标准框架下，所有底层驱动构件放在工程文件夹下的“05_Driver”文件夹中，见第一个规范样例工程“..\ch04-Light\KL25-Light(Component)”下的文件组织。

2）设计构件的最基本要求

这里摘要给出设计构件的最基本要求。

(1) 考虑使用与移植方便。要对构件的共性与个性进行分析，抽取出构件的属性和对外接口函数。希望做到：使用同一芯片的应用系统，构件不更改，直接使用；同系列芯片的同功能底层驱动移植时，仅改动头文件；不同系列芯片的同功能底层驱动移植时，头文件与源程序文件的改动尽可能少。

(2) 要有统一、规范的编码风格与注释。主要涉及文件、函数、变量、宏及结构体类型的命名规范；涉及空格与空行、缩进、断行等的排版规范；涉及文件头、函数头、行及边等的注释规范。具体要求见 5.3.2 节。

(3) 宏的使用限制。宏的使用具有两面性，有提高可维护性一面，也有降低阅读性一面，因此不要随意使用宏。

(4) 不使用全局变量。构件封装时，禁止使用全局变量。

---

① 关于使能(Enable)与禁止(Disable)中断，文献中有多种中文翻译，如使能、开启；除能、关闭等，本书统一使用使能中断与禁止中断术语。

2. 构件封装的前期准备——公共要素文件

我们把同一芯片所有工程均需使用的一些内容放在一个文件中，起个名字叫作"公共要素文件"，具体内容见构件公共要素文件。该文件放在工程文件夹的"..\07_Soft_Component\common"文件夹下，名为 common.h。这里给出基本说明，其他内容见 5.3.3 节，部分内容有所重复，但侧重点不同。

1）KL25 芯片寄存器映射文件

```
#include "MKL25Z4.h"                //包含芯片头文件
```

每个底层驱动构件都是以硬件模块的功能寄存器为操作对象，因此，在 common.h 文件中包含描述芯片寄存器地址映射的头文件，当底层驱动构件引用 common.h 文件时，即可使用片内寄存器映射文件中的定义访问各自相关功能寄存器。

2）位操作宏函数

将编程时经常用到的寄存器位操作，定义成宏函数 BSET、BCLR、BGET 这些容易理解与记忆的标识，表示进行寄存器的置位、清位及获得寄存器某一位状态的操作。BSET、BCLR、BGET 宏定义见 5.3.3 节。

3）重定义基本数据类型

给出基本类型的重定义（别名），有两层含义，一是为了便于移植，一是为了书写方便，见 5.3.3 节。对于构件公共要素文件中的其他内容也将在 5.3.3 节中解释。

4.3.3　KL25 的 GPIO 驱动构件源码及解析

根据构件生产的基本要求设计的第一个构件——GPIO 驱动构件，存放于网上教学资源"..\底层驱动构件\gpio"文件夹，供复制使用，各个工程文件夹下的"..\ Drivers\gpio"文件夹中 GPIO 驱动构件与此一致。

1. GPIO 驱动构件头文件

在 GPIO 驱动构件的头文件（gpio.h）中包含的内容有：头文件说明；防止重复包含的条件编译代码结构"#ifndef… #define… #endif"。用宏定义方式统一了端口号地址偏移量（如 PTA_NUM），为引脚描述量的高 8 位。给出 11 个对外服务函数的接口说明及声明，这些函数包括引脚初始化函数（gpio_init）、设定引脚状态函数（gpio_set）、获取引脚状态函数（gpio_get）三个主要函数，以及反转引脚状态函数（gpio_reverse）、引脚上下拉使能函数（gpio_pull）、使能引脚中断函数（gpio_enable_int）、禁用引脚中断函数（gpio_disable_int）、获取引脚 GPIO 中断状态（gpio_get_int）、清引脚 GPIO 中断（gpio_clear_int）、清所有引脚 GPIO 中断（gpio_clear_allint）、引脚的驱动能力设置函数（gpio_drive_strength）等 8 个功能函数。

```
//===========================================================================
//文件名称：gpio.h
//功能概要：GPIO 底层驱动构件头文件，测试过芯片：KL25、KL26、KW01
//设计单位：苏州大学 NXP 嵌入式中心(sumcu.suda.edu.cn)
//版    本：2012-10-12 V1.0；2016-3-26 V6.0(WYH)
//===========================================================================
```

```
#ifndef _GPIO_H                              //防止重复定义(_GPIO_H 开头)
#define _GPIO_H

#include "common.h"                          //包含公共要素头文件

//端口号地址偏移量宏定义
#define PTA_NUM   (0<<8)
#define PTB_NUM   (1<<8)
#define PTC_NUM   (2<<8)
#define PTD_NUM   (3<<8)
#define PTE_NUM   (4<<8)
//GPIO 引脚方向宏定义
#define GPIO_IN   (0)
#define GPIO_OUTPUT (1)
//GPIO 引脚中断类型宏定义
#define LOW_LEVEL      (8)                   //低电平触发
#define HIGH_LEVEL     (12)                  //高电平触发
#define RISING_EDGE    (9)                   //上升沿触发
#define FALLING_EDGE   (10)                  //下降沿触发
#define DOUBLE_EDGE    (11)                  //双边沿触发

//==========================================================
//函数名称:gpio_init
//函数返回:无
//参数说明:port_pin:(端口号)|(引脚号)(如:(PTB_NUM)|(9) 表示为 B 口 9 号脚)
//         dir:引脚方向(0=输入,1=输出,可用引脚方向宏定义)
//         state:端口引脚初始状态(0=低电平,1=高电平)
//功能概要:初始化指定端口引脚作为 GPIO 引脚功能,并定义为输入或输出,若是输出,
//          还指定初始状态是低电平或高电平
//==========================================================
void gpio_init(uint_16 port_pin, uint_8 dir, uint_8 state);

//==========================================================
//函数名称:gpio_set
//函数返回:无
//参数说明:port_pin:(端口号)|(引脚号)(如:(PTB_NUM)|(9) 表示为 B 口 9 号脚)
//          state:希望设置的端口引脚状态(0=低电平,1=高电平)
//功能概要:当指定端口引脚被定义为 GPIO 功能且为输出时,本函数设定引脚状态
//==========================================================
void gpio_set(uint_16 port_pin, uint_8 state);

//==========================================================
//函数名称:gpio_get
//函数返回:指定端口引脚的状态(1 或 0)
//参数说明:port_pin:(端口号)|(引脚号)(如:(PTB_NUM)|(9) 表示为 B 口 9 号脚)
//功能概要:当指定端口引脚被定义为 GPIO 功能且为输入时,本函数获取指定引脚状态
//==========================================================
uint_8 gpio_get(uint_16 port_pin);
```

```
//==========================================================
//函数名称：gpio_reverse
//函数返回：无
//参数说明：port_pin：(端口号)|(引脚号)(如：(PTB_NUM)|(9) 表示为 B 口 9 号脚)
//功能概要：当指定端口引脚被定义为 GPIO 功能且为输出时，本函数反转引脚状态
//==========================================================
void gpio_reverse(uint_16 port_pin);

//==========================================================
//函数名称：gpio_pull
//函数返回：无
//参数说明：port_pin：(端口号)|(引脚号)(如：(PTB_NUM)|(9) 表示为 B 口 9 号脚)
//          pullselect：下拉/上拉(0=下拉,1=上拉)
//功能概要：当指定端口引脚被定义为 GPIO 功能且为输入时，本函数设置引脚下拉/上拉
//==========================================================
void gpio_pull(uint_16 port_pin, uint_8 pullselect);

//==========================================================
//函数名称：gpio_enable_int
//函数返回：无
//参数说明：port_pin：(端口号)|(引脚号)(如：(PTB_NUM)|(9) 表示为 B 口 9 号脚)
//          irqtype：引脚中断类型,由宏定义给出,再次列举如下：
//                   LOW_LEVEL      8          //低电平触发
//                   HIGH_LEVEL     12         //高电平触发
//                   RISING_EDGE    9          //上升沿触发
//                   FALLING_EDGE   10         //下降沿触发
//                   DOUBLE_EDGE    11         //双边沿触发
//功能概要：当指定端口引脚被定义为 GPIO 功能且为输入时，本函数开启引脚中断,并
//          设置中断触发条件.
//注    意：KL25 芯片,只有 PORTA、PORTD 口具有 GPIO 类中断功能
//==========================================================
void gpio_enable_int(uint_16 port_pin,uint_8 irqtype);

//==========================================================
//函数名称：gpio_disable_int
//函数返回：无
//参数说明：port_pin：(端口号)|(引脚号)(如：(PTB_NUM)|(9) 表示为 B 口 9 号脚)
//功能概要：当指定端口引脚被定义为 GPIO 功能且为输入时，本函数关闭引脚中断
//注    意：KL25 芯片,只有 PORTA、PORTD 口具有 GPIO 类中断功能
//==========================================================
void gpio_disable_int(uint_16 port_pin);

//==========================================================
//函数名称：gpio_get_int
//函数返回：引脚 GPIO 中断标志,1 表示引脚有 GPIO 中断,0 表示引脚无 GPIO 中断
//参数说明：port_pin：(端口号)|(引脚号)(如：(PTB_NUM)|(9) 表示为 B 口 9 号脚)
//功能概要：当指定端口引脚被定义为 GPIO 功能且为输入时,获取中断标志
//注    意：KL25 芯片,只有 PORTA、PORTD 口具有 GPIO 类中断功能
//==========================================================
```

```
uint_8 gpio_get_int(uint_16 port_pin);

//===========================================================
//函数名称：gpio_clear_int
//函数返回：无
//参数说明：port_pin：(端口号)|(引脚号)(如：(PTB_NUM)|(9) 表示为 B 口 9 号脚)
//功能概要：当指定端口引脚被定义为 GPIO 功能且为输入时，清除中断标志
//注　　意：KL25 芯片，只有 PORTA、PORTD 口具有 GPIO 类中断功能
//===========================================================
void gpio_clear_int(uint_16 port_pin);

//===========================================================
//函数名称：gpio_clear_allint
//函数返回：无
//参数说明：无
//功能概要：清除所有端口的 GPIO 中断
//注　　意：KL25 芯片，只有 PORTA、PORTD 口具有 GPIO 类中断功能
//===========================================================
void gpio_clear_allint(void);

//===========================================================
//函数名称：gpio_drive_strength
//函数返回：无
//参数说明：port_pin：(端口号)|(引脚号)(如：(PTB_NUM)|(9) 表示为 B 口 9 号脚)
//        control：控制引脚的驱动能力，1=高驱动，0=正常驱动
//          (引脚被配置为数字输出时，引脚控制寄存器的 DSE=1：高驱动，
//          DSE=0：正常驱动)
//功能概要：(引脚驱动能力：指引脚输入或输出电流的承受力，一般用 mA 单位度量，
//          正常驱动能力 5mA，高驱动能力 18mA.)当引脚被配置为数字输出时，
//          对引脚的驱动能力进行设置，只有 PTB0，PTB1，PTD6，PTD7 同时具有高驱
//          动能力和正常驱动能力，这些引脚可用于直接驱动 LED 或给 MOSFET(金氧
//          半场效晶体管)供电，该函数只适用于上述 4 个引脚
//===========================================================
void gpio_drive_strength(uint_16 port_pin, uint_8 control);

#endif                    //防止重复定义(_GPIO_H 结尾)
```

2. GPIO 驱动构件源程序文件

GPIO 驱动构件的源程序文件(gpio.c)中实现的对外接口函数，主要是对相关寄存器进行配置，从而完成构件的基本功能。构件内部使用的函数也在构件源程序文件中定义。下面给出部分函数的源代码。

```
//===========================================================
//文件名称：gpio.c
//功能概要：GPIO 底层驱动构件源文件
//===========================================================
#include "gpio.h"                              //包含本构件头文件
```

```
//各端口基地址放入常数数据组 PORT_ARR[0]～PORT_ARR[4]中
const PORT_MemMapPtr PORT_ARR[]={PORTA_BASE_PTR,PORTB_BASE_PTR,
                                 PORTC_BASE_PTR,PORTD_BASE_PTR,PORTE_BASE_PTR};
//GPIO 口基地址放入常数数据组 GPIO_ARR[0]～GPIO_ARR[4]中
const GPIO_MemMapPtr GPIO_ARR[]={PTA_BASE_PTR,PTB_BASE_PTR,
                                 PTC_BASE_PTR,PTD_BASE_PTR,PTE_BASE_PTR};

//内部函数声明
void gpio_get_port_pin(uint_16 port_pin,uint_8 * port,uint_8 * pin);

//(函数头注释见头文件)
void gpio_init(uint_16 port_pin, uint_8 dir, uint_8 state)
{
    //局部变量声明
    PORT_MemMapPtr port_ptr;              //声明 port_ptr 为 PORT 结构体类型指针
    GPIO_MemMapPtr gpio_ptr;              //声明 port_ptr 为 GPIO 结构体类型指针
    uint_8 port,pin;                      //声明端口 port、引脚 pin 变量
    //根据带入参数 port_pin,解析出端口与引脚分别赋给 port,pin
    gpio_get_port_pin(port_pin, &port, &pin);
    //根据 port,给局部变量 port_ptr、gpio_ptr 赋值(获得两个基地址)
    port_ptr = PORT_ARR[port];            //获得 PORT 模块相应口基地址
    gpio_ptr = GPIO_ARR[port];            //获得 GPIO 模块相应口基地址

    //设定相应端口的相应引脚功能为 GPIO(即令引脚控制寄存器的 MUX=0b001)
    PORT_PCR_REG(port_ptr, pin) &= ~PORT_PCR_MUX_MASK;    //置 D10-D8=000
    PORT_PCR_REG(port_ptr, pin) |= PORT_PCR_MUX(1);       //置 D10-D8=001

    //根据带入参数 dir,决定引脚为输出还是输入
    if (1 == dir)                         //希望为输出
    {
        BSET(pin,GPIO_PDDR_REG(gpio_ptr)); //数据方向寄存器的 pin 位=1,定义为输出
        gpio_set(port_pin, state);         //调用 gpio_set 函数,设定引脚初始状态
    }
    else                                  //希望为输入
    {
        BCLR(pin,GPIO_PDDR_REG(gpio_ptr)); //数据方向寄存器的 pin 位=0,定义为输入
    }
}

//(函数头注释见头文件)
void gpio_set(uint_16 port_pin, uint_8 state)
{
    //局部变量声明
    GPIO_MemMapPtr gpio_ptr;           //声明 port_ptr 为 GPIO 结构体类型指针(首地址)
    uint_8 port,pin;                   //声明端口 port、引脚 pin 变量
    //根据带入参数 port_pin,解析出端口与引脚分别赋给 port,pin
    gpio_get_port_pin(port_pin, &port, &pin);

    //根据 port,给局部变量 gpio_ptr 赋值(GPIO 基地址)
```

```
    gpio_ptr = GPIO_ARR[port];

    //根据带入参数 state,决定引脚为输出 1 还是 0
    if (1==state)
    {
        BSET(pin,GPIO_PDOR_REG(gpio_ptr));
    }
    else
    {
        BCLR(pin,GPIO_PDOR_REG(gpio_ptr));
    }
}

//(函数头注释见头文件)
uint_8 gpio_get(uint_16 port_pin)
{
    //局部变量声明
    GPIO_MemMapPtr gpio_ptr;              //声明 port_ptr 为 GPIO 结构体类型指针(首地址)
    uint_8 port,pin; //声明端口 port、引脚 pin 变量
    //根据带入参数 port_pin,解析出端口与引脚分别赋给 port,pin
    gpio_get_port_pin(port_pin, &port, &pin);

    //根据 port,给局部变量 gpio_ptr 赋值(GPIO 基地址)
    gpio_ptr = GPIO_ARR[port];

    //返回引脚的状态
    return ((BGET(pin,GPIO_PDIR_REG(gpio_ptr)))>=1 ? 1:0);
}

(限于篇幅,省略其他函数实现,见网上教学资源)

//-------------------------------------以下为内部函数存放处-------------------------------
//=========================================================================
//函数名称: gpio_get_port_pin
//函数返回: 无
//参数说明: port_pin: 端口号|引脚号(如: (PTB_NUM)|(9) 表示为 B 口 9 号脚)
//          port: 端口号(传指带出参数)
//          pin:引脚号(0~31,实际取值由芯片的物理引脚决定)(传指带出参数)
//功能概要: 将传进参数 port_pin 进行解析,得出具体端口号与引脚号
//            分别赋值给 port 与 pin,返回
//           (例: (PTB_NUM)|(9)解析为 PORTB 与 9,并将其分别赋值给 port 与 pin)
//=========================================================================
void gpio_get_port_pin(uint_16 port_pin,uint_8 * port,uint_8 * pin)
{
    * port = (port_pin >> 8);
    * pin = port_pin;
}
//------------------------------------------内部函数结束-----------------------------------
```

3. GPIO 驱动构件源码解析

1）两个结构体类型

在工程文件夹的芯片头文件（“..\CPU\MKL25Z4.h”）中，有端口寄存器结构体，把端口模块的编程寄存器用一个结构体类型（PORT_Type）封装起来：

```
typedef struct {
    __IO uint32_t PCR[32];           //引脚控制寄存器(0～31),偏移：0x0,间隔：0x4
    __O uint32_t GPCLR;              //全局引脚控制寄存器(L),偏移：0x80
    __O uint32_t GPCHR;              //全局引脚控制寄存器(H),偏移：0x84
    uint8_t RESERVED_0[24];          //保留(占位)(0～23)
    __IO uint32_t ISFR;              //中断状态标志寄存器,偏移：0xA0
} PORT_Type, * PORT_MemMapPtr;
```

同时定义了不优化的 PORT 模块寄存器结构体指针（PORT_MemMapPtr），这样，只要给出端口基地址，就可以使用该结构体的成员变量，实现对各寄存器的访问。类似地，给出了 GPIO 模块结构体类型（GPIO_Type）及其指针（GPIO_MemMapPtr）：

```
typedef struct {
    __IO uint32_t PDOR;              //数据输出寄存器,偏移：0x0
    __O uint32_t PSOR;               //输出置 1 寄存器,偏移：0x4
    __O uint32_t PCOR;               //输出清 0 寄存器,偏移：0x8
    __O uint32_t PTOR;               //输出取反寄存器,偏移：0xC
    __I uint32_t PDIR;               //数据输入寄存器,偏移：0x10
    __IO uint32_t PDDR;              //数据方向寄存器,偏移：0x14
} GPIO_Type, * GPIO_MemMapPtr;
```

2）端口模块及 GPIO 模块各口基地址

KL25 的端口（PORT）模块各口基地址：PORTA_BASE_PTR、PORTB_BASE_PTR、PORTC_BASE_PTR、PORTD_BASE_PTR、PORTE_BASE_PTR 在芯片头文件中以宏常数方式给出。KL25 的 GPIO 模块各口基地址 PTA_BASE_PTR、PTB_BASE_PTR、PTC_BASE_PTR、PTD_BASE_PTR、PTE_BASE_PTR 也在芯片头文件中以宏常数方式给出，本程序直接作为指针常量。

3）编程与注释风格

希望仔细分析本构件的编程与注释风格，一开始就规范起来，这样就会逐步锻炼起良好的编程习惯。特别需要注意的是，不要编写令人难以看懂的程序，不要把简单问题复杂化，不要使用不必要的宏。

4.4 利用构件方法控制小灯闪烁

本书用 KL25 控制发光二极管指示灯的例子开始了规范编程的程序之旅，程序中使用了 GPIO 驱动构件来编写指示灯程序。当指示灯两端引脚上有足够高的正向压降时，它就会发光。在本书的工程实例中，小灯采用图 4-2 中的接法。当在 I/O 引脚上输出高或低电

平时，指示灯就会亮或暗。SD-FSL-KL25-EVB硬件板上有个三色灯，分别是PORTB19=红灯、PORTB18=绿灯、PORTB9=蓝灯。按照4.2.3节的方法，使用直接地址编程，放在了网上教学资源的"..\ch04-Light\KL25-Light(Simple)"文件夹下。但提示读者，我们曾说过，直接地址编程方式不妥，应该使用底层驱动构件。

4.4.1　Light构件设计

首先把控制小灯的程序封装成构件"Light"，这个构件是调用芯片底层驱动构件而设计的，我们把调用芯片底层驱动构件设计的面向具体应用的构件，称为应用构件。在工程目录中，把应用构件存放在"06_App_Component"文件夹中。"Light"构件由头文件"Light.h"与源程序文件"light.c"组成。"light.h"就是"Light"构件的使用说明。一个合格的构件头文件应该是一份完备且简明的使用说明，也就是说，不需查看源程序文件就能够完全使用该构件。只有这样，才可以把源程序文件"light.c"变成库文件"liblight.a"。

1. Light构件的头文件light.h

```
//========================================================================
//文件名称：light.h
//功能概要：小灯构件头文件
//设计单位：苏州大学NXP嵌入式中心(sumcu.suda.edu.cn)
//更新记录：2012-02-02 V1.0,2015-02-23 V3.0
//========================================================================

#ifndef _LIGHT_H                              //防止重复定义(_LIGHT_H 开头)
#define _LIGHT_H

//头文件包含
#include "common.h"                           //包含公共要素头文件
#include "gpio.h"                             //用到gpio构件

//指示灯端口及引脚定义
#define LIGHT_RED        (PTB_NUM|19)  //红色RUN灯使用的端口/引脚
#define LIGHT_BLUE       (PTB_NUM|9)   //蓝色RUN灯使用的端口/引脚
#define LIGHT_GREEN      (PTB_NUM|18)  //绿色RUN灯使用的端口/引脚

//灯状态宏定义(灯亮、灯暗对应的物理电平由硬件接法决定)
#define LIGHT_ON         0                    //灯亮
#define LIGHT_OFF        1                    //灯暗

//==================接口函数声明==================
//========================================================================
//函数名称：light_init
//函数参数：port_pin：(端口号)|(引脚号)(如：(PTB_NUM)|(9) 表示为B口9号脚)
//          state：设定小灯状态.由宏定义
//函数返回：无
//功能概要：指示灯驱动初始化
//========================================================================
void light_init(uint_16 port_pin, uint_8 state);
//========================================================================
//函数名称：light_control
```

```
//函数参数：port_pin：(端口号)|(引脚号)(如：(PTB_NUM)|(9) 表示为 B 口 9 号脚)
//          state：设定小灯状态．由宏定义
//函数返回：无
//功能概要：控制指示灯亮暗
//=====================================================================
void light_control(uint_16 port_pin, uint_8 state);

//=====================================================================
//函数名称：light_change
//函数参数：port_pin：(端口号)|(引脚号)(如：(PTB_NUM)|(9) 表示为 B 口 9 号脚)
//函数返回：无
//功能概要：切换指示灯亮暗
//=====================================================================
void light_change(uint_16 port_pin);

#endif //防止重复定义(_LIGHT_H 结尾)
```

2．Light 构件的使用方法

现在，以控制一盏小灯闪烁为例，必须知道两点：一是由芯片的哪个引脚，二是高电平点亮还是低电平点亮。这样就可使用 Light 构件控制小灯了。例如，蓝色小灯由 PTB9 引脚控制，低高电平点亮，使用步骤如下。

（1）在 light.h 文件中给小灯起名字，并确定与 MCU 连接的引脚，进行宏定义。

```
#define  LIGHT_BLUE  (PTB_NUM|9)          //蓝色 RUN 灯使用的端口/引脚
```

（2）在 light.h 文件中对小灯亮、暗进行宏定义，方便编程。

```
#define  LIGHT_ON    0       //灯亮
#define  LIGHT_OFF   1       //灯暗
```

（3）在 main 函数中初始化 LED 灯的初始状态。

```
light_init(LIGHT_BLUE, LIGHT_OFF);            //蓝灯初始化
```

（4）在 main 函数中点亮小灯。

```
light_control(LIGHT_BLUE, LIGHT_ON);            //蓝灯亮
```

这样，MCU 控制一个开关量设备就变得简单而清晰。

3．Light 构件的源程序文件 light.c

这里给出 Light 构件的源程序文件 light.c，供设计应用类构件参考。

```
//=====================================================================
//文件名称：light.c
//功能概要：小灯构件源文件
//=====================================================================
```

```
#include "light.h"

//(函数头注释见头文件)
void light_init(uint_16 port_pin, uint_8 state)
{
    gpio_init(port_pin, GPIO_OUTPUT, state);
}

//(函数头注释见头文件)
void light_control(uint_16 port_pin, uint_8 state)
{
    gpio_set(port_pin, state);
}

//(函数头注释见头文件)
void light_change(uint_16 port_pin)
{
    gpio_reverse(port_pin);
}
```

4.4.2　Light构件测试工程主程序

测试工程位于网上教学资源中的“..\ch04-Light\KL25_Light(Component)”文件夹。功能是SD-FSL-KL25-EVB上的三色灯(红、绿、蓝)闪烁。测试工程主程序如下。

```
//说明见工程文件夹下的Doc文件夹内Readme.txt文件
//==========================================================

#include "includes.h"                    //包含总头文件

int main(void)
{
    //1. 声明主函数使用的变量
    uint_32 mRuncount;                    //主循环计数器
    uint_8 flag;
    //2. 关总中断
    DISABLE_INTERRUPTS;

    //3. 初始化外设模块
    light_init(LIGHT_RED, LIGHT_OFF);     //红灯初始化
    light_init(LIGHT_BLUE, LIGHT_OFF);    //蓝灯初始化
    light_init(LIGHT_GREEN, LIGHT_OFF);   //绿灯初始化

    //4. 给有关变量赋初值
    mRuncount=0;                          //主循环计数器
    flag = 0;                             //灯控制标志
    //5. 使能模块中断
```

```
        //6. 开总中断
        ENABLE_INTERRUPTS;

        //进入主循环
        //主循环开始==================================
        for(;;)
        {
            //三色灯分别闪烁------------------------------------------------------------
            mRuncount++;                        //主循环次数计数器+1
            if (mRuncount >= COUNTER_MAX) //主循环次数计数器大于设定的宏常数
            {
                mRuncount = 0;
                switch(flag)
                {
                case 0: //红灯取反,绿灯暗,蓝灯暗
                    light_change(LIGHT_RED);
                    light_control(LIGHT_BLUE, LIGHT_OFF);
                    light_control(LIGHT_GREEN, LIGHT_OFF);
                    flag = 1;
                    break;
                case 1: //蓝灯取反,红灯暗,绿灯暗
                    light_change(LIGHT_BLUE);
                    light_control(LIGHT_RED, LIGHT_OFF);
                    light_control(LIGHT_GREEN, LIGHT_OFF);
                    flag = 2;
                    break;
                case 2: //绿灯取反,红灯暗,蓝灯暗
                default:
                    light_change(LIGHT_GREEN);
                    light_control(LIGHT_RED, LIGHT_OFF);
                    light_control(LIGHT_BLUE, LIGHT_OFF);
                    flag = 0;
                    break;
                }
            }
            //以下加入用户程序------------------------------------------------------------
        }//主循环 end_for
        //主循环结束==================================
    }
```

其中的常数 COUNTER_MAX,在总头文件中宏定义,决定了小灯的闪烁频率。**这个程序结构已经很清晰,也十分容易理解,接下来运行,但需要开发环境,上述工程是在 KDS 开发环境下组织的,网上教学资源中还给出了其他常用开发环境下的工程组织,但构件及工程框架是不变的。读者可利用开发环境及硬件板进行实际运行、单步调试、模块级调试等。这些工作的基本方法,见网上教学资源中的文档。**

4.5 工程文件组织框架与第一个C语言工程分析

本节以Light工程为例，阐述KDS环境下KL25工程的组织及执行过程。使用其他环境，也使用同样的工程框架。

嵌入式系统工程包含若干文件，包括程序文件、头文件、与编译调试相关的文件、工程说明文件、开发环境生成文件等，文件众多，合理组织这些文件，规范工程组织，可以提高项目的开发效率、提高阅读清晰度、提高可维护性、降低维护难度。工程组织应体现嵌入式软件工程的基本原则与基本思想。这个工程框架也可被称为软件最小系统框架，因为它包含工程的最基本要素。**软件最小系统框架是一个能够点亮一个发光二极管的，甚至带有串口调试构件的，包含工程规范完整要素的可移植与可复用的工程模板。**

4.5.1 工程框架及所含文件简介

图4-3给出以Light工程为例的树状工程结构模板，物理组织与逻辑组织一致。该模板是苏州大学NXP嵌入式中心为在KDS环境下开发ARM Cortex-M4/M0+ Kinetis K/L系列MCU应用工程而设计的。

KL25_Light(Component)	工程名
Binaries	编译链接生成的二进制代码文件
Includes	系统包含文件(自动生成)
01_Doc	<文档文件夹>
02_CPU	<内核相关文件>
03_MCU	< MCU相关文件夹>
MKL25Z4.h	芯片头文件
startup_MKL25Z4.S	启动代码
system_MKL25Z4.c	系统初始化源文件
system_MKL25Z4.h	系统初始化头文件
04_Linker_File	<链接文件夹>
intflash.ld	链接文件
05_Driver	<芯片底层驱动构件文件夹>
gpio	< GPIO底层构件文件夹>
gpio.c	GPIO底层构件源文件
gpio.h	GPIO底层构件头文件
06_App_Component	<应用构件文件夹>
light	<小灯构件文件夹>
light.c	小灯构件源文件
light.h	小灯构件头文件
07_Soft_Component	<软件构件文件夹>
08_Source	<工程主程序文件夹>
includes.h	总头文件
isr.c	中断源文件
main.c	主函数
Debug	<工程输出文件夹>(编译链接自动生成)

图4-3 Light工程的树状工程结构模板

该工程模板与 KDS3.0 提供的 Demo 工程模板相比，简洁易懂，去掉了一些初学者不易理解或不必要的文件，同时应用底层驱动构件化的思想改进了程序结构，重新分类组织了工程，目的是引导读者进行规范的文件组织与编程。

1. 工程名与新建工程

不必在意工程名，而使用工程文件夹标识工程，不同工程文件夹就可区别不同工程。这样工程文件夹内的文件中所含的工程名字不再具有标识意义，可以修改，也可以不修改。建议新工程文件夹使用手动复制标准模板工程文件夹或复制功能更少的旧标准工程的方法来建立，这样，复用的构件已经存在，框架保留，体系清晰。不推荐使用 KDS3.0 或其他开发环境的新建功能来建立一个新工程。

2. 工程文件夹内的基本内容

工程文件夹内编号的共含 8 个下级文件夹，除去 KDS 环境保留的文件夹 Includes 与 Debug，分别是 01_Doc、02_CPU、03_MCU、04_Linker_File、05_Driver、06_App_Component、07_Soft_Component、08_Source。其简明功能及特点见表 4-2。

表 4-2　工程文件夹内的基本内容

编号	文　件　夹	简明功能及特点
1	01_Doc	说明文档文件夹，工程改动时，及时记录
2	02_CPU	CMSIS M0＋内核文件
3	03_MCU	MCU 文件夹，存放芯片头文件及芯片初始化文件，MCU 不同时，芯片头文件需更换
4	04_Linker_File	链接文件夹，放置链接文件
5	05_Driver	底层驱动文件夹，逐步加入各模块驱动构件
6	06_App_Component	存放应用构件。应用构件被定义为通过调用底层驱动构件而完成特定功能的构件，例如 LED、LCD、电机开关构件
7	07_Soft_Component	抽象软件构件文件夹，与硬件不直接相关的软件构件，或调用底层构件完成的功能软件构件
8	08_Source	源程序文件夹，含主程序文件、中断服务例程文件等。这些文件是工程开发人员进行编程的主要对象

3. CPU(内核)相关文件简介

CPU(内核)相关文件(core_cm0plus.h、core_cmFunc.h、core_cmInstr.h)位于工程框架的“..\02_CPU”文件夹内，它们是 ARM 公司提供的符合 CMSIS(Cortex Microcontroller Software Interface Standard，ARM Cortex 微控制器软件接口标准)的内核相关头文件，与供应商无关。其中，core_cm0plus.h 为 ARM Cortex M0＋内核的核内外设访问层头文件，而 core_cmFunc.h 及 core_cmInstr.h 则分别为 ARM Cortex M 系列内核函数及指令访问头文件。使用 CMSIS 标准可简化程序的开发流程，提高程序的可移植性。对任何使用该 CPU 设计的芯片，该文件夹内容相同。

4. MCU(芯片)相关文件简介

MCU(芯片)相关文件(MKL25Z4.h、startup_MKL25Z4.S、system_MKL25Z4.h、system_MKL25Z4.c)位于工程框架的“..\03_MCU”文件夹内。由芯片厂商提供。

芯片头文件 MKL25Z4.h 文件中，给出了芯片专用的寄存器地址映射，设计面向直接硬

件操作的底层驱动时,利用该文件使用映射寄存器名,获得对应地址。该文件一般由芯片设计人员提供,一般嵌入式应用开发者不必修改该文件,只需遵循其中的命名。

启动文件 startup_MKL25Z4.S,包含中断向量表。其分析见4.5.4节。

系统初始化文件 system_MKL25Z4.h、system_MKL25Z4.c,主要存放启动文件 startup_MKL25Z4.S中调用的系统初始化函数SystemInit()及其相关宏常量的定义,此函数实现关闭看门狗及配置系统工作时钟的功能。

5. 应用程序源代码文件——总头文件 includes.h、main.c及中断服务例程文件isr.c

在工程框架的"..\08_Source"文件夹内放置着总头文件 includes.h、main.c及中断服务例程文件isr.c。

总头文件 includes.h是main.c使用的头文件,内含常量、全局变量声明、外部函数及外部变量的引用。

主程序文件 main.c是应用程序的启动后总入口,main函数即在该文件中实现。在main函数中包含一个永久循环,对具体事务过程的操作几乎都是添加在该主循环中。应用程序的执行,一共两条独立的线路,这是一条运行路线。另一条是中断线,在isr.c文件中编程。若有操作系统,则在这里启动操作系统调度器。

中断服务例程文件isr.c是中断处理函数编程的地方,有关中断编程问题将在6.3.3节中阐述。

6. 编译链接产生的其他相关文件简介

映像文件(.map)与列表文件(.lst)位于"..\Debug"文件夹,由编译链接产生。.map文件提供了查看程序、堆栈设置、全局变量、常量等存放的地址信息。.map文件中指定的地址在一定程度上是动态分配的(由编译器决定),工程有任何修改,这些地址都可能发生变动。.lst文件提供了函数编译后,机器码与源代码的对应关系,用于程序分析。

4.5.2　链接文件常用语法及链接文件解析

1. 链接文件的作用

从源代码到最后的可执行文件可以认为需要经过编译、汇编和链接三个过程。每个源代码文件在编译和汇编后都会生成一个可重定位的目标文件(以下简称为中间文件),链接器可以将这些中间文件组合成最终的可执行目标文件(以下简称为目标文件)。如果调用了静态库中的变量或函数的话,链接过程中还会把库文件包括进来。图4-4演示了链接器的作用。

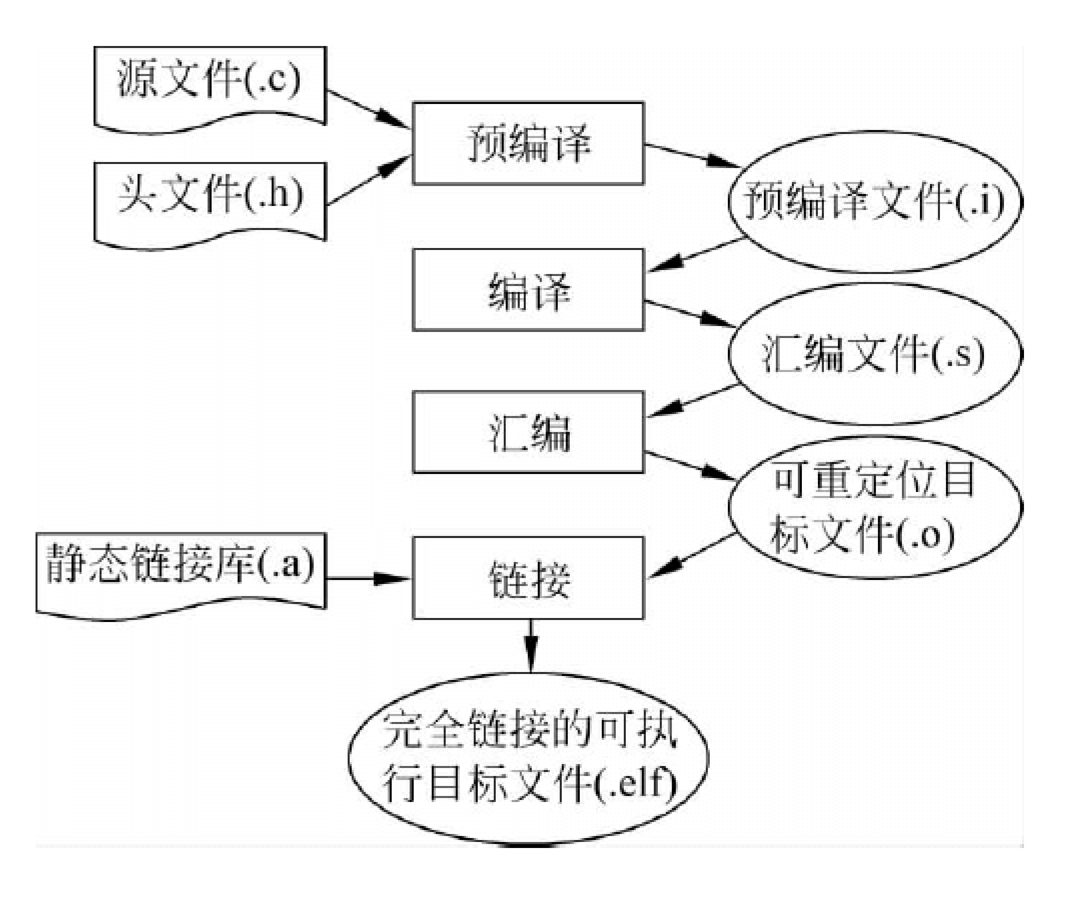


图4-4　程序编译链接的过程

ELF(Executable and Linking Format)文件是一种常用于Linux平台的二进制文件,.o文件通常被叫作对象文件(object file),.o文件是在结构上类似于ELF文件的二进制文件。两者都会被包含一些section

和符号引用。section 用来保存不同数据，比如.text 的 section 用于保存代码，而.data 的 section 则用来存储变量。而符号引用，则是在这个文件中使用的变量或者函数。对于.o 文件，这些符号引用可以是未定义的，因为函数与变量可能存储在别的文件中，ELF 文件中则都应有具体指向的地址。

每一个源文件(.c 文件)经编译汇编后都会生成一个中间文件(.o 文件)。链接器的作用，就是要把中间文件的 section 放到最终可执行文件的合适的 section 中，并且对于符号引用还要找到合适的定义，使得函数调用顺利执行。

04_Linker_File 下的链接文件 intflash.ld 是提供给链接器的链接脚本。链接脚本用于控制链接的过程，规定了如何把输入的中间文件内的 section 放入最终目标文件内，并控制目标文件内各部分的地址分配。

KDS 下，可以在 Properties→C/C++build→Settings 中的 Tool Settings 选项卡下的 Cross ARM C++ Linker 中设置链接脚本的路径与名称。

2. 链接文件常用语法简述

链接脚本文件中常用的命令包括 ENTRY、MEMORY 和 SECTIONS。下面介绍 KL25 链接文件中用到的命令用法[①]。

1）ENTRY 命令

ENTRY 命令的格式如下：

```
ENTRY(SYMBOL)
```

该命令把符号 SYMBOL 的值设置为程序的入口地址。

实际上，此命令在 KL25 的链接过程中并不起作用，在括号内放入任何函数或者删除此语句，程序依然会正常执行。这是因为在 KL25 上，程序总是从中断向量表第一个向量指向的程序开始执行的。在 PC 上的程序需要使用 ENTRY()命令来告诉链接器第一条执行的指令所在的地址，此处保留了这种惯常用法，但是实际并不起作用。

2）MEMORY 命令

MEMORY 命令的格式如下：

```
MEMORY
{
  NAME1[(ATTR)] : ORIGIN = ORIGIN1, LENGTH = LEN1
  NAME2[(ATTR)] : ORIGIN = ORIGIN2, LENGTH = LEN2
...
}
```

[]内的为可选项，有时候不需要。

NAME：存储区域的名字，这个名字可以与符号名、文件名或 SECTION 名重复，因为它处于一个独立的名字空间。

---

① 关于链接器使用方法与链接脚本中支持的命令的完整文档，可以查看 GNU 链接器官方网站上的文档：https://sourceware.org/binutils/docs-2.26/ld/index.html。

ATTR：定义存储区域的属性，在把文件中的 section 输入到目标文件时，如果加上了 ATTR，那么只有输入文件中的 section 符合 ATTR 的值时，才会被输入到输出文件中。ATTR 可用的属性与含义列举在表 4-3 中。

表 4-3 ATTR 可用属性

属 性 名	含 义
R	只读 section
W	读/写 section
X	可执行 section
A	"可分配"的 section
I/L	初始化了的 section
!	不满足该字符后的任何一个属性的 section

ORIGIN：关键字，区域开始地址，可简写成 org 或 o。

LENGTH：关键字，区域的大小，可简写成 len 或 l。

3）SECTIONS 命令

SECTIONS 的命令格式如下：

```
SECTIONS
{
SECTIONS_COMMAND
SECTIONS_COMMAND
...
}
```

常用的 SECTIONS_COMMAND 命令包括符号赋值语句和输出 section 描述的语句。

符号赋值语句比较容易理解，类似于 C 语言中的赋值语句，将一个常量的值赋给一个符号。这个符号可以在链接文件中使用，也可以被链接过程中的中间文件所引用。

输出 section 描述语句则是 SECTIONS 命令的核心，该语句用于确定最后输出的目标文件中的 section 应该由哪些中间文件的哪些 section 来构成，此语句的常用语法如下：

```
SECTIONS
SECTION : [AT(LMA)]
{
OUTPUT-SECTION-COMMAND
OUTPUT-SECTION-COMMAND
...
} [> REGION]
```

[]中的为可选项，可以不使用。

SECTION 是最后的目标文件中该 section 使用的名称，比如. interrupts，. text 或者. data。

AT(LMA)可以指定该 section 保存的地址，LMA 是一个地址，可以是具体的数值，也可以是保存有地址的符号常量。

> REGION 则用于把该 section 输出到指定的存储区域中，REGION 是存储区域的名称，应该在 MEMORY 命令中定义。

OUTPUT-SECTION-COMMAND 是具体的输出命令，可以控制输出到最后目标文件中的 section，还可以定义一些符号常量。下面以 KL25 工程中. interrupts 为例解析一下 OUTPUT-SECTION-COMMAND 的用法。

```
SECTIONS
{
...
  .interrupts :
  {
    __VECTOR_TABLE = .;
    . = ALIGN(4);
    KEEP( * (.isr_vector))
    . = ALIGN(4);
  } > m_interrupt
...
}
```

. interrupts 是这个 section 的名称，其后跟的冒号和大括号是必需的，指定了这个 section 的描述范围。

__VECTOR_TABLE = .：这个语句中的“.”是一个特殊的符号，可以理解为一个位置的指针，指向的是当前的地址位置。此语句就是把当前的地址赋给__VECTOR_TABLE 符号。

. = ALIGN(4)：把当前地址做 4 字节的对齐。

KEEP(* (.isr_vector))：KEEP()命令用于告诉链接器，链接时需要保留的 section，不能过滤掉。这是因为在链接输入时，链接器可能将某些认为不需要的 section 过滤掉。 * 是一个通配符，代表所有输入的中间文件。(.isr_vector)是指向中间文件中名字为 isr_vector 的 section。上面已经提到，链接需要把多个中间文件链接成一个目标文件。所以，这个语句会把所有输入的中间文件中的名字为 isr_vector 的 section，输出到目标文件中的名字叫 interrupts 的 section。

接着“> m_interrupt”语句的意思是指把 interrupts 这个 section 放到在 MEMORY 命令中定义的名为 m_interrupt 的存储区域中。

3. KL25 工程中链接文件分析

表 4-4 给出了 KL25-Light 工程的 intflash. ld 文件的简要分析，分析中只抽取与程序、数据安排相关部分说明，其他部分略去。这个文件在同一芯片的所有工程中，原则上不改变。

表 4-4　链接文件简明分析

顺序/命令	内　容	简 要 说 明
(1) 指明程序开始点	ENTRY(Reset_Handler)	程序开始点。这里的 Reset_Handler 是指“03_MCU”中的 startup_MKL25Z4. S 文件中的 Reset_Handler 函数(只是沿用 PC 链接文件格式，在这里本句无意义)

续表

顺序/命令	内　容	简要说明
(2) 设置堆栈的大小	HEAP_SIZE=DEFINED(__heap_size__) ? __heap_size__ : 0x0400; STACK_SIZE = DEFINED(__stack_size__) ? __stack_size__ : 0x0400;	设置堆栈的大小,默认空间大小都是 0x0400(KB)。如果用户设置了__heap_size__或__stack_size__符号值,则使用这两个值作为堆栈的大小
(3) MEMORY,定义和划分存储空间可用的资源	m_interrupts (RX) : ORIGIN = 0x00000000, LENGTH = 0x00000100	中断向量区,256B
	m_flash_config (RX) : ORIGIN = 0x00000400, LENGTH = 0x00000010	Flash 配置区,长度为 16B
	m_text (RX) : ORIGIN = 0x00000800, LENGTH = 0x0001F7FF	Flash 用户程序区,跳过保留中断向量后,开始存放程序、常量。这里地址从 0x800 开始而不是接上面的 0x410,是因为 Flash 章中的加密解密功能需要擦除第一扇区,所以第一扇区不能保存数据。如果不用到 Flash 的加密解密,此处可以改为 ORIGIN = 0x00000410, LENGTH = 0x0001FBF0
	m_data (RW) : ORIGIN = 0x1FFFF000, LENGTH = 0x00004000	RAM 数据区。定义两个数据存储区,分别在 RAM 的 SRAM_L、SRAM_H 区域。整个数据区的长度为 16KB,就是 RAM 的大小
(4) SECTIONS,对各个块的内容的定义	.interrupts : …	中断向量表,在 ROM 中存放在.interrupts 段,起始地址 0x00000000
	.flash_config: …	Flash 配置域相关参数存放在 flash_config 段中,起始地址为 0x00000400
	.text : …	代码段。程序存放在 Falsh 存储区中的.text 段。文件中的 *(.text)指程序,*(.rodata)指常量
	.data: …	标准数据段,可以用来初始化全局变量和静态变量
	.bss: …	未初始化全局变量和静态变量段,在.data 之后
(5) SECTIONS 对栈相关符号赋值	__StackTop = ORIGIN(m_data) + LENGTH(m_data); ___StackLimit = ___StackTop-STACK_SIZE;	给__StackTop 符号赋上值,此符号会在“03_MCU”中的 startup_MKL25Z4.S 文件中的中断向量表中使用到。__StackLimit 是栈的最小地址,小于此地址则会导致栈溢出

4.5.3　机器码文件解析

KDS 开发平台,针对 KL 系列 MCU,使用 Cross ARM GCC 编译器,在编译链接过程中生成针对 ARM CPU 的.elf 格式可执行代码,同时也可生成十六进制(.hex)格式的机器码。

.elf(Executable and Linking Format,可执行链接格式)最初由 UNIX 系统实验室

(UNIX System Laboratories，USL)作为应用程序二进制接口(Application Binary Interface，ABI)的一部分而制定和发布。其最大特点在于它有比较广泛的适用性，通用的二进制接口定义使之可以平滑地移植到多种不同的操作环境上。UltraEdit 软件工具可查看.elf 文件内容。

.hex(Intel HEX)文件是由一行行符合 Intel HEX 文件格式的文本所构成的 ASCII 文本文件，在 Intel HEX 文件中，每一行包含一个 HEX 记录，这些记录由对应机器语言码(含常量数据)的十六进制编码数字组成。在 KDS 环境下，直接双击并按 F5 键刷新可查看该文件。

1. 记录格式

.hex 文件中有 6 种不同类型的语句，但总体格式是一样的，根据表 4-5 格式来记录。

表 4-5　.hex 文件记录行语义

	字段 1	字段 2	字段 3	字段 4	字段 5	字段 6
名称	记录标记	记录长度	偏移量	记录类型	数据/信息区	校验和
长度	1 字节	1 字节	2 字节	1 字节	N 字节	1 字节
内容	开始标记“:”		数据类型记录有效；非数据类型，该字段为“0000”	00-数据记录；01-文件结束记录；02-扩展段地址；03-开始段地址；04-扩展线性地址；05-链接开始地址	取决于记录类型	开始标记之后字段的所有字节之和的补码。校验和＝0xFF－(记录长度＋记录偏移＋记录类型＋数据段)＋0x01

2. 实例分析

以 KL25_Light(Component)工程中的 KL25_Light(Component).hex 为例，进行简明分析。截取第一个实例工程的“.hex”文件的部分行进行分解，见表 4-6。

表 4-6　KL25_Light(Component).hex 文件部分行分解

行	记录标记	记录长度	偏移量	记录类型	数据/信息区	校验和
1	:	10	0000	00	00300020010800004508000045080000	FD
2	:	10	0010	00	00000000000000000000000000000000	E0
12	:	10	00B0	00	45080000450800004508000045080000	0C
14	:	10	0800	00	72B600F0CDF962B609490A4A0A4B9B1A	42
102	:	00	0000	01		FF

(1) 分析第 1 行，“:10000000003000200108000045080000450800000FD”。进行语义分割“:10 0000 00 00300020 01080000 4508000045080000FD”，分析如下：以“:”开始，长度为“0x10”(16 个字节)，“0000”表示偏移量(含义见表 4-5)，紧接着的“00”代表记录类型为数据类型(含义见表 4-5，00 表示数据记录)，接下来的就是数据段“00300020 010800004508000045080000”，表示该数据段是存放在偏移地址为“0000”的存储区的机器操作码，也就是说，只有这些数据被写入到 Flash 存储区。值得注意的是，这里的 hex 文件中，数据部分是以“小端”的方式存储的，这与 MCU 内部的存储方式有关。在这种格式中，字的

低字节存储在低地址中，而字的高字节存放在高地址中，第1个字(4个字节)是“00 30 00 20”，实际表示的数据内容为“20 00 30 00”，就是堆栈栈顶(=RAM最高地址+1)，参见链接文件与映像文件，这4个字节也就是中断向量表中开始内容(占用了0号中断位置)，其内容由MCU内部机制在MCU启动时被放入堆栈寄存器SP中。接下来的一个字(4个字节)“01 08 00 00”→“00 00 08 01”，占用中断向量表1号中断位置(即复位向量)，该数减1，在MCU启动时被放入程序计数器PC中，那么就从存储器的0x00000800地址(见第14行)中取出指令，开始执行程序了。从源程序角度，即开始执行复位中断处理程序Reset_Handler。可以从“.map”文件、“.lst”文件找到相应信息进行理解，例如，此时Reset_Handler的地址为0x00000800。至于为什么文件中的0x00000800，到了机器码中变成了0x00000801，这是因为Cortex-M0+处理器的指令地址为半字对齐，也就意味着PC寄存器的最低位必须始终为0。但是，程序在跳转时，PC的最低位必须被置为1，以表明内核仍然处于thumb状态。而不是ARM状态，参见《ARM-Cortex-M0权威指南》。

(2) 从第1行后半部开始至12行，是中断向量表，每4个字节代表一个中断向量号。

(3) 第13行是Flash配置域，被保留，未进行赋值。

(4) 第102行(最后一行)为文档的结束记录，记录类型为“0x01”；“0xFF”为本记录的校验和字段内容。

综合分析工程的.map文件、.ld文件、.hex文件、.lst文件，可以理解程序的执行过程，也可以对生成的机器码进行分析对比。

4.5.4　芯片上电启动运行过程解析

1. 链接文件中对中断向量表的定位

工程文件夹中“..\04_Linker_File\intflash.ld”文件中MEMORY部分，有下列语句：

```
m_interrupts (RX) : ORIGIN = 0x00000000, LENGTH = 0x00000100
```

该语句确定了m_interrupts代表从0x00000000开始，长度为0x00000100。在随后的SECTIONS部分有下列语句：

```
.interrupts :
{
  _VECTOR_TABLE = .;
  . = ALIGN(4);
  KEEP(*(.isr_vector))                    /* Startup code */
  . = ALIGN(4);
} > m_interrupts
```

这样*(.isr_vector)指向0x00000000地址。相关内容可以在“Debug”文件夹下的.map文件中可以找到，中断的起始地址是从0x00000000开始，长度为0x00000100。

2. 启动文件startup_MKL25Z4.S解析

启动文件“..\03_MCU\startup_MKL25Z4.S”中包含中断向量表及启动代码。

中断向量表是指按照中断源的中断向量号(见表 4-6)中的固定顺序,存放中断服务程序入口地址的一段存储区域。每个中断服务程序入口地址占用 4 个字节单元,KL25/26 中断向量表的位置在存储区 0x0000_0000～0x0000_00BC 的一段地址范围,一共 48×4=192 个字节,存放 48 个中断服务程序的入口地址。中断服务程序的入口地址又称为中断向量或中断向量指针,它指向中断服务程序在存储器中的位置。

如何将中断服务程序的入口地址(中断向量),按照中断源的中断向量序号的顺序,写入到中断向量表的位置呢?就是从给定的地址放常数而已,工程编译时由链接文件 intflash.ld 指定这组常数存放的首地址。这里给出 startup_MKL25Z4.S 文件的部分内容(表 4-7)。

表 4-7　启动文件 startup_MKL25Z4.S 解析

内　容	解　析
.section .isr_vector, "a" .align 2 .globl __isr_vector __isr_vector:	(1) 定义中断向量表全局数组名__isr_vector,与链接文件 intflash.ld 中指定的区域.isr_vector 关联。这里标号"__isr_vector:"就是 intflash.ld 中"isr_vector"①。即是地址:0x0000_00000
.long __StackTop /* Top of Stack */ .long Reset_Handler /* Reset Handler */ … … .long UART2_IRQHandler /* UART2 */ … … .long PORTD_IRQHandler /* PORTD Pin */ .size __isr_vector, . - __isr_vector	(2) 为中断向量表的所有表项填入默认值,即以中断向量所对应外设的英文名作为中断处理函数的函数名。0x0000_00000～0x0000_00003 地址填写的__StackTop 在 intflash.ld 中可以找到,是 0x20003000②,0x0000_00004～0x0000_00007 地址填写 Reset_Handler(复位处理程序函数名)。这两个区域属于特殊用途。随后各区域填写对应的默认中断处理函数的函数名。例如,在串口 2 模块的中断向量表项里填入 UART2_IRQHandler③
/* Flash 配置 */ .section .FlashConfig, "a" .long 0xFFFFFFFF .long 0xFFFFFFFF .long 0xFFFFFFFF .long 0xFFFFFFFE	(3) 给出 Flash 配置。主要用于芯片加密,参见第 9 章

① 可以在链接文件 intflash.ld 中看出,isr_vector 指向 0x0000_00000 地址。

② 是堆栈栈顶,是芯片内 RAM 最大地址+1。该芯片堆栈方向是向小地址使用的,因此,栈顶设在 RAM 最大地址+1。堆栈空间是临时变量的空间。全局变量从小地址向大地址顺序使用,这样两头向中间使用,符合使用规则。

③ 这里把 Handler 翻译成"处理程序",也可以翻译成"句柄",就是中断服务程序的入口地址,也就是中断服务程序的函数名。

续表

<table>
<tr><th>内　容</th><th>解　析</th></tr>
<tr><td><pre>/* Reset_Handler 入口 */
    .thumb_func
    .align 2
    .globl Reset_Handler
    .weak Reset_Handler
    .type Reset_Handler, %function
Reset_Handler:
    cpsid i   /* 关总中断 */
#ifndef _NO_SYSTEM_INIT
    bl SystemInit   /* 调用系统初始化函数 */
#endif
    cpsie i   /* 开总中断 */
    …
    以下工作:
    把数据从 ROM 复制到 RAM 中
    给未初始化的变量赋初值“0”
    转到 main 函数</pre></td><td>(4) 给出复位处理程序 Reset_Handler 的代码实现。内容是:关总中断、调用系统初始化函数 SystemInit(在 system_MKL25Z4.c 中)、把数据从 ROM 复制到 RAM 中、给未初始化的变量赋初值“0”、调用 main 函数(即转到 main 函数运行了,main 函数是个永久循环,就一直在那里运行各种操作了,这就是启动过程)</td></tr>
<tr><td><pre>DefaultISR:
    ldr r0, =DefaultISR
    bx   r0</pre></td><td>(5) 实现一个默认处理函数 Default_Handler,其内容为一个永久循环。实际应用程序可以修改这个内容,以便进行特殊处理</td></tr>
<tr><td><pre>   .macro def_irq_handler handler_name
    .weak \handler_name
    .set \handler_name, DefaultISR
    .endm
    def_irq_handler   NMI_Handler
    def_irq_handler   HardFault_Handler
    …
    def_irq_handler   UART2_IRQHandler
    …
    def_irq_handler   PORTD_IRQHandler

    .end</pre></td><td>(6) 以弱符号[①]的方式,将默认中断处理函数的函数名指向默认处理函数 Default_Handler。实际使用时,只需在中断服务程序文件 isr.c 中再定义一个与所需中断处理函数的函数名同名函数即可。例如,UART2_IRQHandle{};其中函数名 UART2_IRQHandler 与此处相同,此时编译器默认将其识别为强符号,在编译时会覆盖掉这里的以弱符号定义的默认值。到此,中断向量表得以实现。有关中断编程问题,转入第 6 章讨论</td></tr>
</table>

这里再对弱符号进行一些说明。看下列语句:

```
    .macro   def_irq_handler handler_name
    .weak    \handler_name
    .set     \handler_name, Default_Handler
    .endm
```

使用.macro 指令,定义一个宏,可以把需要重复运行的一段代码或者一组指令缩写成一个宏,在程序调用的时候可以直接去调用这个宏而使代码更加简洁清晰。这里定义的宏为“def_irq_handler”(读作“默认中断处理程序”),代表的是后面的“handler_name”(读作

① 弱符号可被同名强符号覆盖,C 语言中编译器默认函数和初始化了的全局变量为强符号。

"处理程序名")，这相当于是一个宏函数。使用"弱定义.weak"来定义"handler_name"，用户如果重写了"handler_name"对应的中断处理程序，将会覆盖这里给出的对应默认中断处理程序，若不使用"弱定义.weak"，重写对应中断处理程序，编译器会认为是重复定义，将会报错。灵活使用"弱定义.weak"，能避免不少烦琐。随后的.set 语句将跳转到"Default_Handler"程序段，如下：

```
DefaultISR:
    ldr r0, =DefaultISR
    bx r0
    .size DefaultISR, .-DefaultISR.
```

这部分默认的中断处理例程的宏。默认的处理只是一个无限循环，用户重写相应的处理例程后就可以覆盖默认的处理方式。

接下来是一系列宏定义：

```
def_irq_handler    NMI_Handler
    ...
```

这一系列中断处理被宏定义为 def_irq_handler，大大缩减了代码量，当用户在 isr.c 文件中重新定义后，会覆盖相应的中断服务例程，提高了程序的健壮性和可复用性。

3. main 函数之前的程序运行过程简明流程总结

芯片复位到 main 函数之前程序运行过程总结如下。

(1) 芯片上电复位后，芯片内部机制首先从 Flash 的 0x00000000 地址中，取出第一个表项的内容，赋给内核寄存器 SP(堆栈指针)，完成堆栈指针初始化。例如，在链接文件 intflash.ld 中，链接时将_StackTop 的值放到 Flash 的 0x00000000 地址中。

(2) 芯片内部机制将第二个表项的内容，赋给内核寄存器 PC(程序计数器)。

(3) 由于该表项存放启动函数 Reset_Handler 的首地址，因而运行"..\03_MCU\startup_MKL25Z4.S"文件中的 Reset_Handler 函数，关闭了总中断，运行"..\03_MCU\system_MKL25Z4.c"文件中的 SystemInit()函数，进行芯片部分初始化设置，依次关闭看门狗、系统时钟初始化，再回到 Reset_Handler 中继续剩余初始化功能，包括开总中断、将 ROM 中的初始化数据复制到 RAM 中、清零未初始化 BSS 数据段、随后进入用户主函数 main。

实际应用中，可根据是否启动看门狗、是否复制中断向量表至 RAM、是否清零未初始化 BSS 数据段等要求来修改此文件。初学者，在未理解相关内容的情况下，不建议修改 startup_MKL25Z4.S 及 system_MKL25Z4.c 文件内容。主函数 main 中的无限循环之前的代码顺序，有一定规范，将在第 6 章以第一个带中断的程序为例。

4.6　第一个汇编语言工程：控制小灯闪烁

汇编语言编程给人的第一种感觉就是难，相对于 C 语言编程，汇编在编程的直观性、编程效率以及可读性等方面都有所欠缺，但掌握基本的汇编语言编程方法是嵌入式学习的基

本功,可以增加嵌入式编程者的“内力”。

在本书教学资料中提供的KDS开发环境中,汇编程序是通过新建工程的方式并且经过修改芯片初始化组织起来的。汇编工程通常包含芯片相关的程序框架文件、软件构件文件、工程设置文件、主程序文件及抽象构件文件等。下面将结合第一个KL25汇编工程实例“KL25_Light(asm)”,讲解上述的文件概念,并详细分析KL25汇编工程的组成、汇编程序文件的编写规范、软硬件模块的合理划分等。读者若能认真分析与实践第一个汇编实例程序,可以达到由此入门的目的。

4.6.1　汇编工程文件的组织

汇编工程的样例在“..\ch04-Light\KL25_Light(asm)”文件夹中。本汇编工程类似C工程,仍然按构件方式进行组织。图4-5给出了小灯闪烁汇编工程的树状结构,主要包括MCU相关头文件夹、底层驱动构件文件夹、Debug工程输出文件夹、程序文件夹等。读者可按照理解C工程的方式,理解这个结构。

KL25_Light(asm)	工程名
Binaries	编译链接生成的二进制代码文件
Includes	系统包含文件(自动生成)
01_Doc	<文档文件夹>
02_CPU	<内核相关文件>
03_MCU	< MCU相关文件夹>
MKL25Z4.h	KL25芯片头文件
startup_MKL25Z4.S	启动代码
system_MKL25Z4.c	系统初始化源文件
system_MKL25Z4.h	系统初始化头文件
04_Linker_File	<链接文件夹>
intflash.ld	链接文件
05_Driver	<芯片底层驱动构件文件夹>
gpio	< GPIO底层构件文件夹>
gpio.S	GPIO底层构件源文件
gpio.inc	GPIO底层构件头文件
06_App_Component	<应用构件文件夹>
light	<小灯构件文件夹>
light.S	小灯构件源文件
light.inc	小灯构件头文件
07_Soft_Component	<软件构件文件夹>
common	<通用代码文件夹>
08_Source	<工程主程序文件夹>
include.S	总头文件
main.S	主函数
Debug	<工程输出文件夹>(编译链接自动生成)

图4-5　小灯闪烁汇编工程的树状结构

汇编工程仅包含一个汇编主程序文件,该文件名固定为main.s。汇编程序的主体是程序的主干,要尽可能简洁、清晰、明了,程序中的其余功能,尽量由子程序去完成,主程序主要完成对子程序的循环调用。主程序文件main.s,包含以下内容。

(1) 工程描述：工程名、程序描述、版本、日期等。若调试过程中有新的体会，也可在此添加。目的是为将来自己使用，或为同组开发提供必要的备忘信息。

(2) 总头文件：声明全局变量和包含主程序文件中需要的头文件、宏定义等。

(3) 主程序：主程序一般包括初始化与主循环两大部分。初始化包括堆栈初始化、系统初始化、I/O 端口初始化、中断初始化等。主循环是程序的工作循环，根据实际需要安排程序段，但一般不宜过长，建议不要超过 100 行，具体功能可通过调用子程序来实现，或由中断程序实现。

(4) 内部直接调用子程序：若有不单独存盘的子程序，建议放在此处。这样在主程序总循环的最后一个语句就可以看到这些子程序。每个子程序不要超过 100 行。若有更多的子程序请单独存盘，单独测试。

(5) 外部子程序：若程序使用独立存盘的子程序，可使用"#include"包含。比如该主函数 main 中需要用到控制小灯闪烁频率的延时数 RUN_COUNTER_MAX，而该参数是在 include.s 头文件中定义的，所以需要在 main 函数中把该头文件用 include 包含在内，即 #include "include.s"。

4.6.2　汇编语言 GPIO 构件及使用方法

汇编语言 GPIO 构件包含头文件 gpio.inc 及汇编源程序文件 gpio.s，功能与 C 语言 GPIO 构件一致。

1. GPIO 构件的头文件 gpio.inc

```
#=========================================================================
#文件名称：gpio.inc
#功能概要：KL25 GPIO 底层驱动构件(汇编)头文件
#版权所有：苏州大学 NXP 嵌入式中心(sumcu.suda.edu.cn)
#版本更新：2013-06-05 V1.0；2016-03-03 V2.0
#=========================================================================
#端口号地址偏移量宏定义
.equ PTA_NUM,(0<<8)
.equ PTB_NUM,(1<<8)
.equ PTC_NUM,(2<<8)
.equ PTD_NUM,(3<<8)
.equ PTE_NUM,(4<<8)
#GPIO 引脚方向宏定义
.equ GPIO_IN,(0)
.equ GPIO_OUTPUT,(1)
#GPIO 引脚中断类型宏定义
.equ LOW_LEVEL,(8)                    @低电平触发
.equ HIGH_LEVEL,(12)                  @高电平触发
.equ RISING_EDGE,(9)                  @上升沿触发
.equ FALLING_EDGE,(10)                @下降沿触发
.equ DOUBLE_EDGE,(11)                 @双边沿触发

#引脚控制寄存器基地址宏定义(只给出 PORTA 的引脚控制寄存器 PCR0 的地址,其他由此计算)
```

```
.equ PORT_PCR_BASE,0x40049000 @PORTA的引脚控制寄存器PCR0的地址
#GPIO寄存器基地址宏定义(只给出PORTA的输出寄存器PDOR的地址,其他由此计算)
.equ PORT_GPIO,0x400ff000 @PORTA的输出寄存器PDOR的地址

#=====================================================================
# 函数名称:gpio_init
# 函数返回:无
# 参数说明:r0:(端口号|(引脚号)),例:(PTB_NUM|(5u))表示B口5脚,头文件中有宏定义
#          r2:引脚方向(0=输入,1=输出,可用引脚方向宏定义)
#          r3:端口引脚初始状态(0=低电平,1=高电平)
# 功能概要:初始化指定端口引脚作为GPIO引脚功能,并定义为输入或输出,若是输出,
#          还指定初始状态是低电平或高电平
# 备   注:端口x的每个引脚控制寄存器地址=PORT_PCR_BASE+x*0x1000+n*4
#          其中: x=0~4,对应A~E; n=0~31
#=====================================================================
(其他函数略)
```

2. GPIO构件的汇编源程序gpio.s

```
#=====================================================================
#文件名称:gpio.s
#功能概要:KL25 GPIO底层驱动构件(汇编)程序文件
#=====================================================================

#include "gpio.inc"

#=====================================================================
# 函数名称:gpio_init
# 函数返回:无
# 参数说明:r0:(端口号|(引脚号)),例:(PTB_NUM|(5u))表示B口5脚,头文件中有宏定义
#          r2:引脚方向(0=输入,1=输出,可用引脚方向宏定义)
#          r3:端口引脚初始状态(0=低电平,1=高电平)
# 功能概要:初始化指定端口引脚作为GPIO引脚功能,并定义为输入或输出.若是输出,
#          还指定初始状态是低电平或高电平
# 备   注:端口x的每个引脚控制寄存器地址=PORT_PCR_BASE+x*0x1000+n*4
#          其中:x=0~4,对应A~E;n=0~31
#=====================================================================
gpio_init:
    push {r0-r7,lr}                  @保存现场,pc(lr)入栈
    #-------------------------------------------------------------------
    #从入口参数r0中解析出端口号引脚号,分别放在r0和r1中
    bl gpio_port_pin_resolution      @调用内部解析函数,r0=端口号,r1=引脚号
    #获得待操作端口的第一个PCR寄存器的地址
    mov r7,r0                        @r7=r0=端口号
    ldr r4,=0x1000                   @r4=各端口基地址差值(0x1000)
    mul r7,r7,r4                     @r7=待操作端口与A口的偏移地址
    ldr r4,=PORT_PCR_BASE            @r4=端口A的PCR基地址(即PORT_PCR_BASE)
    add r7,r4                        @r7=待操作端口的第一个PCR寄存器的地址
    #获得待操作引脚PCR寄存器的地址
```

```
    mov r4,r1                @r4=r1=引脚号
    mov r5,#4                @各引脚的 PCR 寄存器地址之间差为 0x04
    mul r4,r4,r5             @r4=待操作引脚 PCR 寄存器的偏移地址
    add r7,r4                @r7=待操作引脚 PCR 寄存器的地址
    #待操作引脚 PCR 寄存器的 MUX 位(10-8 位)清 0
    ldr r4,=0xfffff8ff
    ldr r5,[r7]              @r5=待操作引脚 PCR 寄存器中的内容
    and r5,r4                @待操作引脚 PCR 寄存器的 MUX 字段清零,其余位不变
    #待操作引脚 PCR 寄存器的 MUX 位(10-8 位)置 001,即设置为 GPIO 功能
    ldr r4,=0x00000100
    orr r5,r4                @或运算设 MUX=001,引脚被配置为 GPIO 功能
    str r5,[r7]              @将 r5 中的 MUX 值更新到待操作引脚 PCR 寄存器中
    #求待操作 GPIO 口的基地址(也就是 PDOR 的地址)
    ldr r4,=PORT_GPIO        @r4=PORTA 基地址(GPIO 的基地址)
    mov r7,r0                @r7=r0=端口号
    mov r6,#0x40             @r6=各 GPIO 口基地址差值(40h)
    mul r6,r6,r7             @r6=待操作 GPIO 口的地址偏移
    add r4,r6                @r4=待操作 GPIO 口的地址,也就是 PDOR 的地址

    #根据入口参数 r3,通过对 PDOR 的编程,设置相应引脚为低电平或者高电平
    mov r6,#1
    lsl r6,r6,r1             @r6=待操作的 PDOR 掩码(为 1 的位由 r1 决定)
    cmp r3,#1
    bne gpio_init_1          @r3≠1 转 gpio_init_1,r3=1 继续执行
    #r3=1,设置 PDOR 相应位为 1
    ldr r5,[r4]              @r5=PDOR 中内容
    orr r5,r6                @或运算设置 PDOR 相应位为 1
    str r5,[r4]              @将 r5 中的值更新到待操作端口 PDOR 寄存器中
    bl gpio_init_2
gpio_init_1:
    #r3=0,设置 PDOR 相应位为 0
    mvn r6,r6                @r6 进行取反,即 0 变 1,1 变 0
    ldr r5,[r4]              @r5=PDOR 中内容
    and r5,r6                @与运算设置 PDOR 相应位为 0
    str r5,[r4]              @将 r5 中的值更新到待操作端口 PDOR 寄存器中
gpio_init_2:
    add r4,#0x14             @r4=待操作 GPIO 口 PDDR 寄存器的地址
    #根据入口参数 r2,通过对 PDDR 进行编程,确定引脚为输入或者输出(0 为输入,1 为输出)
    mov r6,#1
    lsl r6,r6,r1             @r6=待操作引脚的 PDDR 掩码(为 1 的位由 r1 决定)
    cmp r2,#1
    bne gpio_init_3          @r2≠1 转 gpio_init_3,r2=1 继续执行
    #r2=1,设置 PDDR 相应位为 1
    ldr r5,[r4]              @r5=PDDR 中内容
    orr r5,r6                @或运算设置 PDDR 相应位为 1
    str r5,[r4]              @将 r5 中的值更新到待操作端口 PDDR 寄存器中
    bl gpio_init_4
gpio_init_3:
    #r2=0,设置 PDDR 相应位为 0
    mvn r6,r6                @r6 进行取反,即 0 变 1,1 变 0
    ldr r5,[r4]              @r5=PDDR 中内容
```

```
    and r5,r6              @与运算设置 PDDR 相应位为 0
    str r5,[r4]            @将 r5 中的值更新到待操作端口 PDDR 寄存器中
gpio_init_4:
    #-------------------------------------------------------------------------
    pop {r0-r7,pc}         @恢复现场,lr 出栈到 pc(即子程序返回)

    (其他函数略)
```

4.6.3 汇编语言 Light 构件及使用方法

汇编语言 Light 构件包含头文件 light.inc 及汇编源程序文件 light.s,用于控制指示灯的亮或暗。包括小灯初始化程序 light_init、控制小灯亮暗程序 light_control 以及切换小灯亮暗程序 light_change。

1. Light 构件的头文件 light.inc

```
#==============================================================
#文件名称: light.inc
#功能概要: 小灯驱动程序文件
#==============================================================

#include "gpio.S"
#指示灯端口及引脚定义
.equ LIGHT_RED,(PTB_NUM|19)              @红色 RUN 灯使用的端口/引脚
.equ LIGHT_BLUE,(PTB_NUM|9)              @蓝色 RUN 灯使用的端口/引脚
.equ LIGHT_GREEN,(PTB_NUM|18)            @绿色 RUN 灯使用的端口/引脚

#灯状态宏定义(灯亮、灯暗对应的物理电平由硬件接法决定)
.equ LIGHT_ON,0                          @灯亮
.equ LIGHT_OFF,1                         @灯暗
```

2. Light 构件的汇编源程序 light.s

```
#==============================================================
#文件名称: light.s
#功能概要: 小灯驱动程序文件
#==============================================================
#include "light.inc"

#==============================================================
# 函数名称: light_init
# 函数返回: 无
# 参数说明: r0:(端口号)|(引脚号),例:(PTB_NUM|(5u))表示 B 口 5 脚,头文件中有宏定义
#           r3:设定小灯状态。由 light.inc 中宏定义
# 功能概要: 指示灯驱动初始化
#==============================================================
light_init:
    push {r0-r3,lr}                @保存现场,将下一条指令地址入栈
```

```
        mov r2, #1                 @小灯为输出
        bl gpio_init               @调用 gpio 初始化函数
        pop {r0-r3, pc}            @恢复现场,返回主程序处继续运行

#===========================================================================
# 函数名称: light_control
# 函数返回: 无
# 参数说明: r0: (端口号)|(引脚号),例:(PTB_NUM|(5u))表示 B 口 5 脚,头文件中有宏定义
#           r3: 设定小灯状态。由 light.inc 中宏定义
# 功能概要: 控制指示灯亮暗
#===========================================================================
light_control:
        push {r0-r3, lr}
        mov r2, #1                 @小灯为输出
        bl gpio_set                @调用 gpio 引脚设置函数
        pop {r0-r3, pc}

#===========================================================================
# 函数名称: light_change
# 函数返回: 无
# 参数说明: r0:(端口号)|(引脚号),例:(PTB_NUM|(5u))表示 B 口 5 脚,头文件中有宏定义
# 功能概要: 切换指示灯亮暗
#===========================================================================
light_change:
        push {r0-r3, lr}
        bl gpio_reverse            @调用后 sgpio 引脚反转函数
        pop {r0-r3, pc}
```

3. 汇编语言 Light 构件的使用方法

现在,以控制一盏小灯闪烁为例,读者必须知道两点:一是由芯片的哪个引脚,二是高电平点亮还是低电平点亮。这样就可使用 Light 构件控制小灯了。例如,小灯由 PTB9 引脚控制,高电平点亮,使用步骤如下。

(1) 在 light.inc 文件中给小灯起名字,并确定与 MCU 连接的引脚,进行宏定义;

```
.equ LIGHT_BLUE,(PTB_NUM|9)      @蓝色 RUN 灯使用的端口/引脚
```

(2) 在 light.inc 文件中对小灯亮、暗进行宏定义,方便编程:

```
equ LIGHT_ON,0           @灯亮
equ LIGHT_OFF,1          @灯暗
```

(3) 在 main 函数中初始化 LED 灯的初始状态:

```
ldr r0,=RUN_LIGHT_BLUE        @r0 指明端口和引脚(用=是因为宏常数>=256,且用 ldr)
mov r3,#LIGHT_OFF             @r3 指明引脚的初始状态
bl light_init                 @调用小灯初始化函数
```

(4) 在 main 函数中点亮小灯：

```
bl light_change           @相等,则调用小灯亮暗转变函数
```

4.6.4 汇编语言 Light 测试工程主程序及汇编工程运行过程

1. Light 测试工程主程序

因为该工程中需要调用 Light 构件的接口函数，所以在 include.s 文件中需要包含 light.s。首先调用 light_Init 函数，初始化所需的指示灯。注意初始化时，要让每一盏灯初始状态为"暗"。随后，通过 light_control 函数控制指示灯亮、暗。通过变量的递增并且设置频率后，就能够在程序运行中，可以比较明显地看到指示灯对应的小灯进行闪烁的现象。

```
#===========================================
#文件名称: main.s
#功能概要: 汇编编程控制小灯闪烁
#版权所有: 苏州大学恩智浦嵌入式中心(sumcu.suda.edu.cn)
#版本更新: 2016-03-27
#===========================================
#include "include.S"

#start 主函数定义开始
    .section .text.main
    .global main                @定义全局变量,在芯片初始化之后调用
    .align 2                    @指令对齐
    .type main function         @定义主函数类
    .align 2
#end 主函数定义结束
main:
    cpsid i                     @关闭总中断
    #小灯初始化, r0,r3 是 light_init 的入口参数
    ldr r0,=RUN_LIGHT_BLUE      @r0 指明端口和引脚(用=是因为宏常数≥256,且用 ldr)
    mov r3,#LIGHT_OFF           @r3 指明引脚的初始状态
    bl light_init               @调用小灯初始化函数
    cpsie i                     @开总中断
#主循环开始====================================
main_loop1:
    ldr r4,=RUN_COUNTER_MAX @取延时值到 r4
    mov r5,#0                   @从零计数
loop:
    add r5,#1                   @加 1 计数
    cmp r4,r5                   @r4 值与 r5 值比较
    bne loop                    @不相等,则跳转 loop
    bl light_change             @相等,则调用小灯亮暗转变函数
    bne main_loop1              @跳转 main_loop1
#主循环结束====================================
.end
```

2. 汇编工程运行过程

当KL25芯片内电复位或热复位后，系统程序的运行过程可分为两部分：main函数之前的运行和main函数之后的运行。

其中，mian函数之前的运行过程和4.5节C语言控制小灯闪烁的运行过程一样，所以具体的过程可以参考4.5节加以体会和理解。

下面对于main函数之后的运行进行简要分析。

首先进入main函数后先对所用到的模块进行初始化，比如小灯端口引脚的初始化，然后进入main_loop1函数，在该函数中首先把一个延时数RUN_COUNTER_MAX存储到寄存器r4中，该延时数用于控制小灯的闪烁频率，可在单步调试中把它改成小值。接着使寄存器r5从零开始递增，每次加1并且同时和寄存器r4中的值比较，如果两个寄存器中的值相等，则调用小灯亮暗转变函数，然后继续运行main_loop1，否则寄存器r5的值继续递增直到和r4寄存器中的值相等为止。

最后当某个中断发生后，MCU将转到中断向量表文件isr.asm所指定的中断入口地址处开始运行中断服务程序（Interrupt Service Routine，ISR），因为该小灯程序没有中断向量表文件，所以此处就不再描述汇编中断程序，深入学习的读者，不难完成此任务。

小　　结

本章作为全书的重点和难点之一，给出了MCU的C语言工程编程框架，对第一个C语言入门工程进行了较为详尽的阐述。透彻理解工程的组织原则、组织方式及运行过程，对后续的学习将有很大的铺垫作用。

(1) GPIO是输入/输出的最基本形式，MCU的引脚若作为GPIO输入引脚，即开关量输入，其含义就是MCU内部程序可以获取该引脚的状态，是高电平1，或是低电平0。若作为输出引脚，即开关量输出，其含义就是MCU内部程序可以控制该引脚的状态，是高电平1，或是低电平0。希望掌握开关量输入/输出电路的基本连接方法。

(2) 本章通过点亮一盏小灯的过程来开启嵌入式学习之旅。为了能够理解直接与硬件打交道的底层原理，4.2节简明扼要给出了KL的端口控制模块与GPIO模块的编程结构，举例通过给映像寄存器赋值的方法，点亮一盏小灯的编程步骤，以便理解底层驱动的含义与编程方法。重点掌握引脚控制寄存器的MUX位段（D10～D8位），是通过该位段确定引脚实际功能的，例如MUX＝0b001，确定引脚为GPIO功能。对于GPIO编程，理解4.2.3节给出的GPIO基本编程步骤。关键是，进行的实际编程与单步调试，理解如何通过对MCU内部寄存器的编程，实现干预MCU引脚的基本过程，这样就可理解软件如何与硬件密切联系。

(3) 为了一开始就进行规范编程。4.3节给出了GPIO驱动构件封装方法与驱动构件封装规范简要说明。在实际工程应用中，为了提高程序的可移植性，不能在所有的程序中都直接操作对应的寄存器，需要将对底层的操作封装成构件，对外提供接口函数，上层只需在调用时传进对应的参数即可完成相应功能，具体封装时用.c文件保存构件的实现代码用.h文件保存需对外提供的完整函数信息及必要的说明。4.3节中给出了GPIO构件的设计方

法。在GPIO构件中设计了引脚初始化(gpio_init)、设定引脚状态(gpio_set)、获取引脚状态(gpio_get)、反转引脚状态(gpio_reverse)、使能引脚中断(gpio_enable_int)、禁用引脚中断(gpio_disable_int)等函数,使用这些接口函数可基本完成对GPIO引脚的操作。4.4节给出了利用GPIO构件,设计操作小灯的应用构件Light。

(4) 嵌入式系统工程往往包含许多文件,有程序文件、头文件、与编译调试相关的文件、工程说明文件、开发环境生成文件等,合理组织这些文件规范工程组织可以提高项目的开发效率和可维护性,工程组织应体现嵌入式软件工程的基本原则与基本思想。本书提供的工程框架主要包括01_Doc、02_CPU、03_MCU、04_Linker_File、05_Driver、06_App_Component、07_Soft_Component、08_Source共8个文件夹,每个文件夹下存放不同功能的文件,通过文件夹的名称可直接体现出来,用户今后在使用时无须新建工程,复制后改名即为新工程。4.5节给出了这些文件夹的功能说明。实际编程工作,在08_Source文件夹中进行,总头文件includes.h是main.c使用的头文件,内含常量、全局变量声明、外部函数及外部变量的引用。主程序文件main.c是应用程序的启动后总入口,main函数即在该文件中实现。在main函数中包含一个永久循环,对具体事务过程的操作几乎都是添加在该主循环中。应用程序的执行,一共有两条独立的线路,这是一条运行路线。另一条是中断线,在isr.c文件中编程。若有操作系统,则在这里启动操作系统调度器。中断服务例程文件isr.c是中断处理函数编程的地方。

(5) 4.6节给出了一个规范的汇编工程样例,供汇编入门使用,读者可以实际调试理解该样例工程,达到初步理解汇编语言编程的目的。对于嵌入式初学者来说,理解一个汇编语言程序是十分必要的。

习　题

1. 举例给出使用对直接映像地址赋值的方法,实现对一盏小灯编程控制的程序语句。

2. 在第一个样例程序的工程组织图中,哪些文件是由用户编写的?哪些是由开发环境编译链接产生的?

3. 简述第一个样例程序的运行过程。

4. 给出.ld文件的功能要点。

5. 参考Light构件设计一个"Button"构件,实现获得拨码开关状态的功能,在编码过程中遵循嵌入式设计编码基本规范。

6. 说明全局变量在哪个文件声明,在哪个文件中给全局变量中赋初值,举例说明一个全局变量的存放地址。

7. 综合分析.hex文件、.map文件、.lst文件,在第一个样例工程中找出system_MKL25Z4.c文件中SystemInit函数、main.c文件中main函数的存放地址,给出各函数前16个机器码,并找到其在.hex文件中的位置。

8. 自行完成一个汇编工程,功能、难易程度自定。

第 5 章　嵌入式硬件构件与底层驱动构件基本规范

本章导读：本章主要分析嵌入式系统构件化设计的重要性和必要性，给出嵌入式硬件构件概念及嵌入式硬件构件的分类、基于嵌入式硬件构件的电路原理图设计简明规则；给出嵌入式底层驱动构件的概念与层次模型；给出底层驱动构件的封装规范，包括构件设计的基本思想与基本原则、编码风格基本规范、头文件及源程序设计规范；给出硬件构件及底层软件构件的重用与移植方法。本章的目的是期望通过一定的规范，提高嵌入式软硬件设计的可重用性和可移植性。

5.1　嵌入式硬件构件

机械、建筑等传统产业的运作模式是先生产符合标准的构件（零部件），然后将标准构件按照规则组装成实际产品。其中，构件（Component）是核心和基础，复用是必需的手段。传统产业的成功充分证明了这种模式的可行性和正确性。软件产业的发展借鉴了这种模式，为标准软件构件的生产和复用确立了举足轻重的地位。

随着微控制器及应用处理器内部 Flash 存储器可靠性提高及擦写方式的变化，内部 RAM 及 Flash 存储器容量的增大以及外部模块内置化程度的提高，嵌入式系统的设计复杂性、设计规模及开发手段已经发生了根本变化。在嵌入式系统发展的最初阶段，嵌入式系统硬件和软件设计通常是由一个工程师来承担，软件在整个工作中的比例很小。随着时间的推移，硬件设计变得越来越复杂，软件的分量也急剧增长，嵌入式开发人员也由一人发展为由若干人组成的开发团队。为此希望提高软硬件设计可重用性与可移植性，构件的设计与应用是重用与移植的基础与保障。

5.1.1　嵌入式硬件构件概念与嵌入式硬件构件分类

要提高硬件设计可重用性与可移植性，就必须有工程师们共同遵守的硬件设计规范。设计人员若凭借个人工作经验和习惯的积累进行系统硬件电路的设计，在开发完一个嵌入式应用系统再进行下一个应用开发时，硬件电路原理图往往需要从零开始，重新绘制；或者在一个类似的原理图上修改，但往往又很麻烦，容易出错。因此把构件的思想引入到硬件原理图设计中。

1. 嵌入式硬件构件概念

什么是嵌入式硬件构件？它与我们常说的硬件模块有什么不同？

众所周知，嵌入式硬件是任何嵌入式产品不可分割的重要组成部分，是整个嵌入式系统的构建基础，嵌入式应用程序和操作系统都运行在特定的硬件体系上。一个以 MCU 为核心的嵌入式系统通常包括以下硬件模块：电源、写入器接口电路、硬件支撑电路、UART、

USB、Flash、AD、DA、LCD、键盘、传感器输入电路、通信电路、信号放大电路、驱动电路等模块。其中有些模块集成在 MCU 内部,有的位于 MCU 之外。

与硬件模块的概念不同,嵌入式硬件构件是指将一个或多个硬件功能模块、支撑电路及其功能描述封装成一个可重用的硬件实体,并提供一系列规范的输入/输出接口。由定义可知,传统概念中的硬件模块是硬件构件的组成部分,一个硬件构件可能包含一个硬件功能模块,也有可能包含多个。

2. 嵌入式硬件构件分类

根据接口之间的生产消费关系,接口可分为供给接口和需求接口两类。根据所拥有接口类型的不同,硬件构件分为核心构件、中间构件和终端构件三种类型。核心构件只有供给接口,没有需求接口。也就是说,它只为其他硬件构件提供服务,而不接受服务。在以单 MCU 为核心的嵌入式系统中,MCU 的最小系统就是典型的核心构件。中间构件既有需求接口又有供给接口,即它不仅能够接受其他构件提供的服务,而且也能够为其他构件提供服务。而终端构件只有需求接口,它只接受其他构件提供的服务。这三种类型构件的区别如表 5-1 所示。

表 5-1　核心构件、中间构件和终端构件的区别

类　　型	供给接口	需求接口	举　　例
核心构件	有	无	芯片的硬件最小系统
中间构件	有	有	电源控制构件、232 电平转换构件
终端构件	无	有	LCD 构件、LED 构件、键盘构件

利用硬件构件进行嵌入式系统硬件设计之前,应该进行硬件构件的合理划分,按照一定规则,设计与系统目标功能无关的构件个体,然后进行“组装”,完成具体系统的硬件设计。这样,这些构件个体也可以被组装到其他嵌入式系统中。在硬件构件被应用到具体系统时,绘制电路原理图阶段,设计人员需要做的仅仅是为需求接口添加接口网标[①]。

5.1.2　基于嵌入式硬件构件的电路原理图设计简明规则

在绘制原理图时,一个硬件构件使用一个虚线框(见图 5-1～图 5-4),把硬件构件的电路及文字描述括在其中,对外接口引出到虚线框之外,填上接口网标。

1. 硬件构件设计的通用规则

在设计硬件构件的电路原理图时,需遵循以下基本原则。

(1) 元器件命名格式:对于核心构件,其元器件直接编号命名,同种类型的元件命名时冠以相同的字母前缀。如电阻名称为 R1、R2 等,电容名称为 C1、C2 等,电感名称为 L1、L2 等,指示灯名称为 E1、E2 等,二极管名称为 D1、D2 等,三极管名称为 Q1、Q2 等,开关名称为 K1、K2 等。对于中间构件和终端构件,其元器件命名格式采用“构件名-标志字符?”。例如,LCD 构件中所有的电阻名称统一为“LCD-R?”,电容名称统一为“LCD-C?”。当构件原理图应用到具体系统中时,可借助原理图编辑软件为其自动编号。

① 电路原理图中网标是指一种连线标识名字,凡是网标相同的地方,表示是连接在一起的。与此对应的,还有一种标识,就是文字标识,仅仅是一种注释说明,不具备电路连接功能。

(2) 为硬件构件添加详细的文字描述，包括中文名称、英文名称、功能描述、接口描述、注意事项等，以增强原理图的可读性。中英文名称应简洁明了。

(3) 将前两步产生的内容封装在一个虚线框内，组成硬件构件的内部实体。

(4) 为该硬件构件添加与其他构件交互的输入/输出接口标识。接口标识有两种：接口注释和接口网标。它们的区别是：接口注释标于虚线框以内，是为构件接口所做的解释性文字，目的是帮助设计人员在使用该构件时，理解该接口的含义和功能；而接口网标位于虚线框之外，且具有电路连接特性。为使原理图阅读者便于区分，接口注释采用斜体字。

在进行核心构件、中间构件和终端构件的设计时，除了要遵循上述的通用规则外，还要兼顾各自的接口特性、地位和作用。

2. 核心构件设计规则

设计核心构件时，需考虑的问题是："核心构件能为其他构件提供哪些信号?"核心构件其实就是某型号 MCU 的硬件最小系统。**核心构件设计的目标是：凡是使用该 MCU 进行硬件系统设计时，核心构件可以直接"组装"到系统中，无须任何改动。**为了实现这一目标，在设计核心构件的实体时必须考虑细致、周全，包括稳定性、扩展性等，封装要完整。核心构件的接口都是为其他构件提供服务的，因此接口标识均为接口网标。在进行接口设计时，需将所有可能使用到的引脚都标注上接口网标(不要考虑核心构件将会用到怎样的系统中去)。若同一引脚具有不同功能，则接口网标依据第一功能选项命名。遵循上述规则设计核心构件的好处是：当使用核心构件和其他构件一起组装系统时，只要考虑其他构件将要连接到核心构件的哪个接口(不是考虑核心构件将要连接到其他构件的哪个接口)，这也符合设计人员的思维习惯。附录 B 给出的 KL25 硬件最小系统原理图就是核心构件的一个典型实例。

3. 中间构件设计规则

设计中间构件时，需考虑的问题是："中间构件需要接收哪些信号，以及提供哪些信号?"中间构件是核心构件与终端构件之间通信的桥梁。在进行中间构件的实体封装时，实体的涉及范围应从构件功能和编程接口两方面考虑。一个中间构件应具有明确的且相对独立的功能，它既要有接收其他构件提供的服务的接口，即需求接口，又要有为其他构件提供服务的接口，即供给接口。描述需求接口采用接口注释，处于虚线框内，描述供给接口采用接口网标，处于虚线框外。

中间构件的接口数目没有核心构件那样丰富。为直观起见，设计中间构件时，将构件的需求接口放置在构件实体的左侧，供给接口放置在右侧。接口网标的命名规则是：构件名称-引脚信号/功能名称。而接口注释名称前的构件名称可有可无，它的命名隐含相应的引脚功能。

电源控制构件(如图 5-1 所示)、可变频率产生构件(如图 5-2 所示)是常用的中间构件。图 5-1 中的 *Power-IN* 和图 5-2 中的 *SDI*、*SCK* 和 *SEN* 均为接口注释，Power-OUT 和 LTC6903-OUT 为接口网标。

4. 终端构件设计规则

设计终端构件时，需考虑的问题是："终端构件需要什么信号才能工作?"终端构件是嵌入式系统中最常见的构件。终端构件没有供给接口，它仅有与上一级构件交付的需求接口，

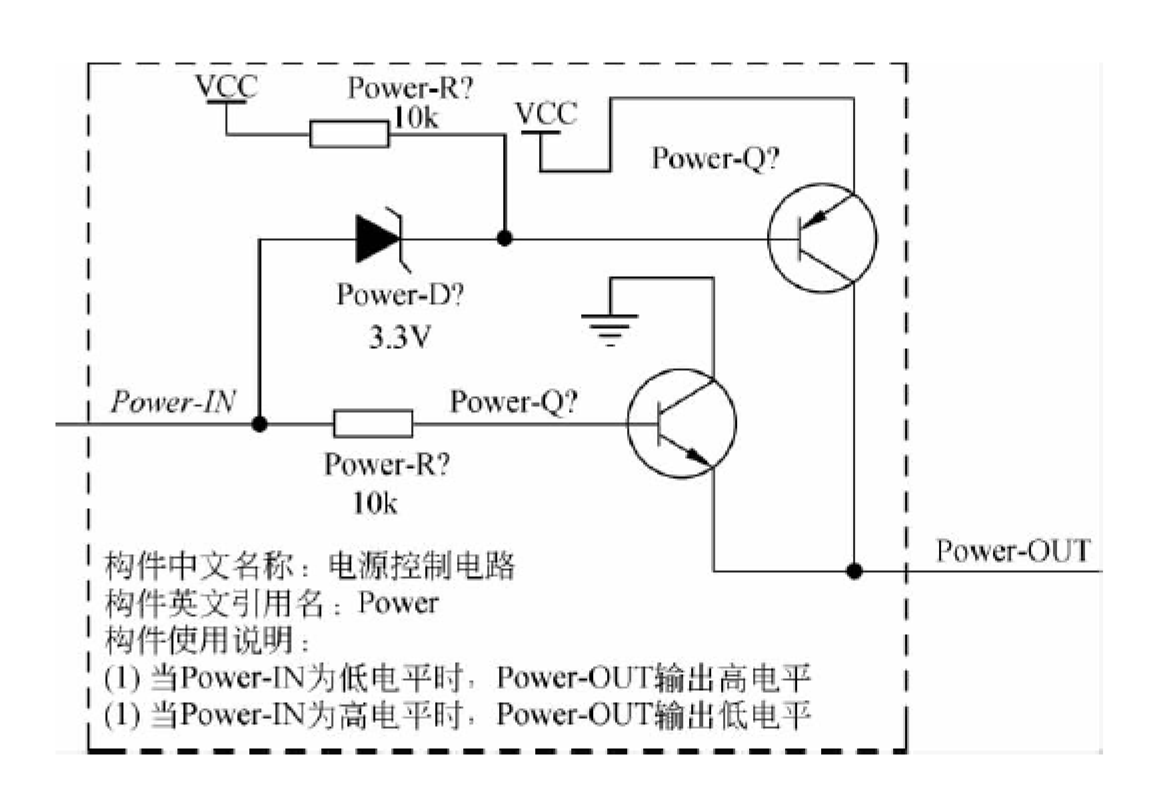


图 5-1　电源控制构件

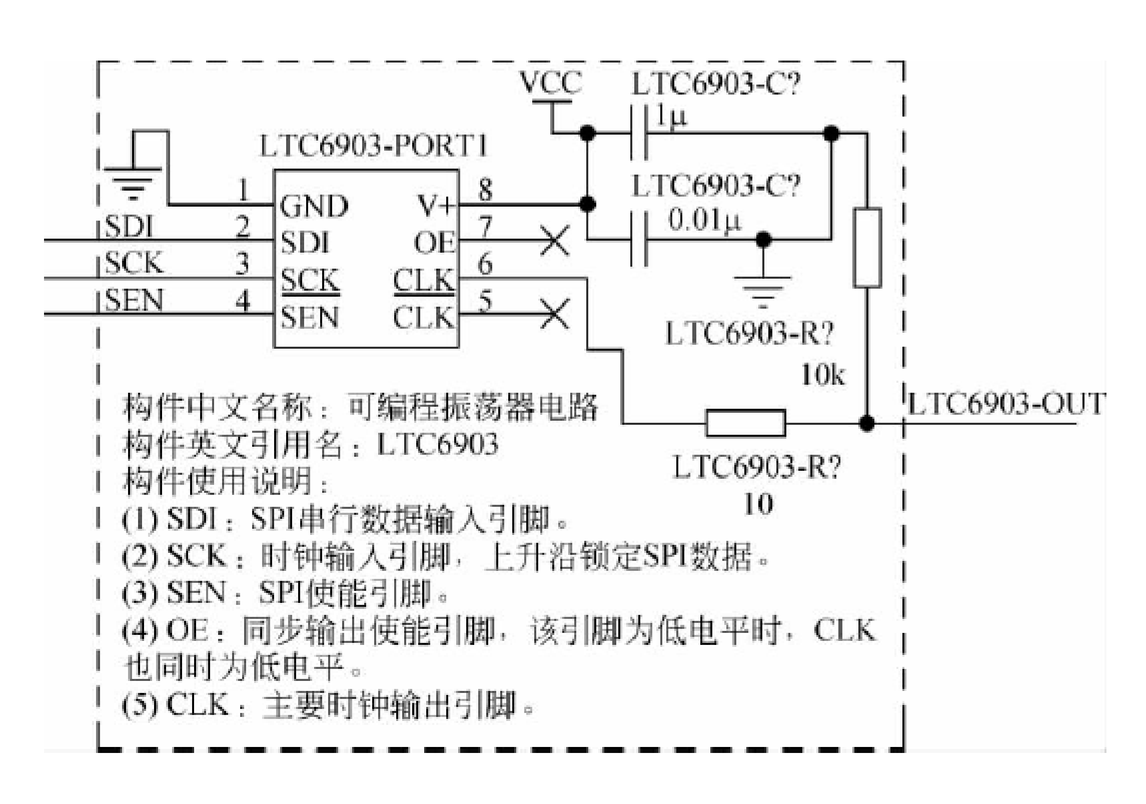


图 5-2　可变频率产生构件

因而接口标识均为斜体标注的接口注释。LCD(YM1602C)构件(如图 5-3 所示)、LED 构件、指示灯构件以及键盘构件(如图 5-4 所示)等都是典型的终端构件。

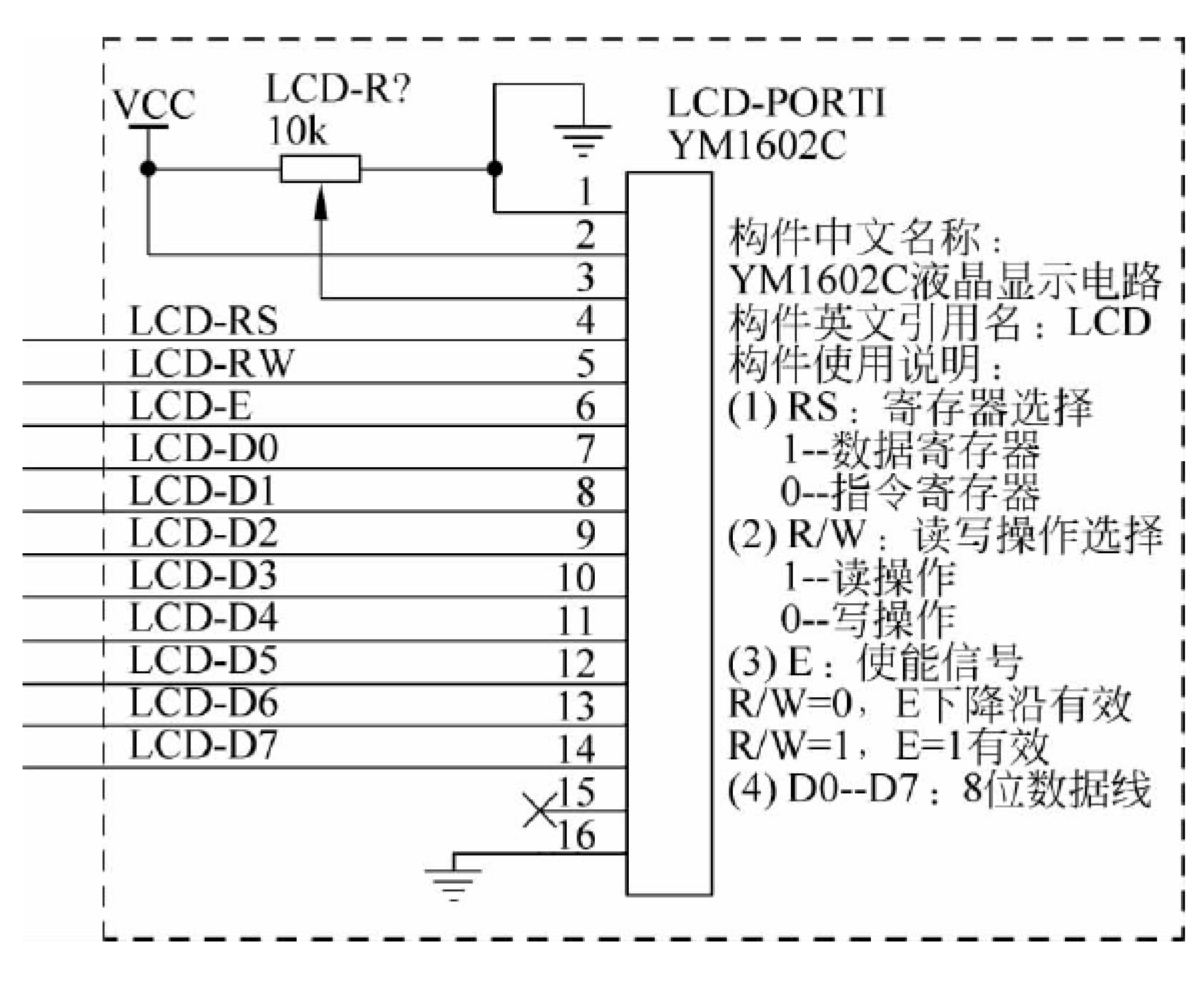


图 5-3　LCD 构件

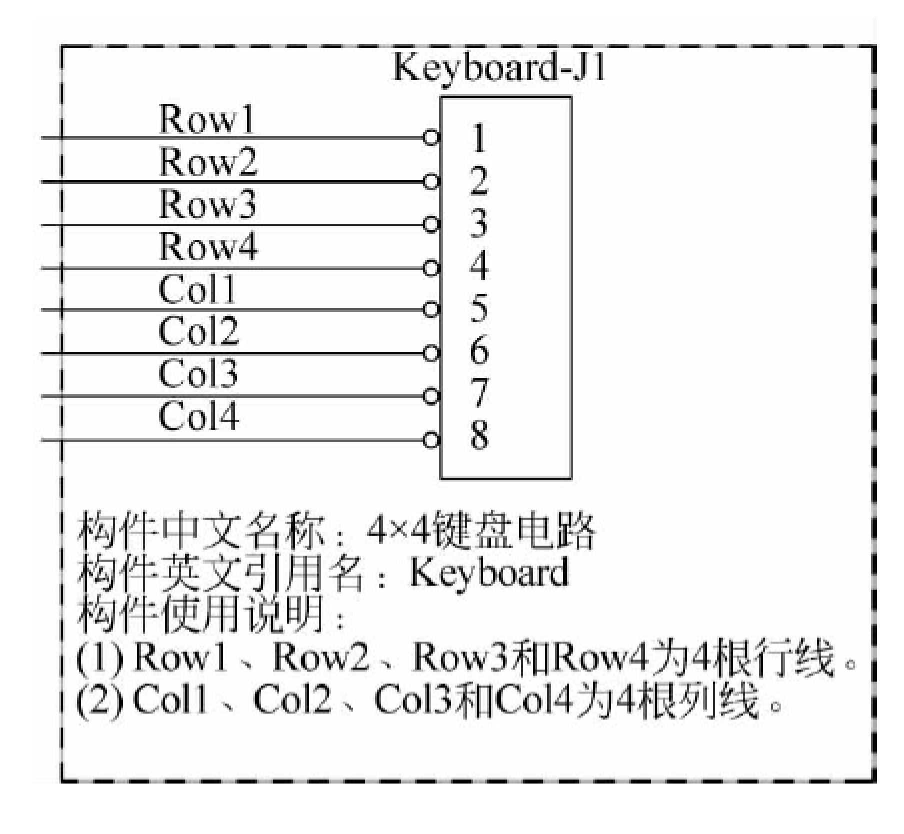


图 5-4　键盘构件

5. 使用硬件构件组装系统的方法

对于核心构件，在应用到具体的系统中时，不必作任何改动。具有相同 MCU 的应用

系统，其核心构件完全相同。对于中间构件和终端构件，在应用到具体的系统中时，仅需为需求接口添加接口网标，在不同的系统中，接口网标名称不同，但构件实体内部完全相同。

使用硬件构件化思想设计嵌入式硬件系统的过程与步骤如下。

(1) 根据系统的功能划分出若干个硬件构件。

(2) 将所有硬件构件原理图“组装”在一起。

(3) 为中间构件和终端构件添加接口网标。

5.2　嵌入式底层驱动构件的概念与层次模型

嵌入式系统是软件与硬件的综合体，硬件设计和软件设计相辅相成。嵌入式系统中的驱动程序是直接工作在各种硬件设备上的软件，是硬件和高层软件之间的桥梁。正是通过驱动程序，各种硬件设备才能正常运行，达到既定的工作效果。

5.2.1　嵌入式底层驱动构件的概念

要提高软件设计可重用性与可移植性，就必须充分理解和应用软件构件技术。“提高代码质量和生产力的唯一最佳方法就是**复用**好的代码”，软件构件技术是软件复用实现的重要方法，也是软件复用技术研究的重点。

构件(Component)是可重用的实体，它包含合乎规范的接口和功能实现，能够被独立部署和被第三方组装①。

软件构件(Software Component)是指，在软件系统中具有相对独立功能、可以明确辨识构件实体。

嵌入式软件构件(Embedded Software Component)是实现一定嵌入式系统功能的一组封装的、规范的、可重用的、具有嵌入特性的软件构件单元，是组织嵌入式系统功能的基本单位。嵌入式软件分为高层软件构件和底层软件构件(底层驱动构件)。高层软件构件与硬件无关，例如，实现嵌入式软件算法的算法构件、队列构件等。而底层驱动构件与硬件密不可分，是硬件驱动程序的构件化封装。下面给嵌入式底层驱动构建一个简明定义。

嵌入式底层驱动构件，简称底层驱动构件或硬件驱动构件，是直接面向硬件操作的程序代码及使用说明。规范的底层驱动构件由头文件(.h)及源程序文件(.c)文件构成②，头文件(.h)应该是底层驱动构件简明且完备的使用说明，也就是说，不需查看源程序文件情况下，就能够完全使用该构件进行上一层程序的开发。为此，设计底层驱动构件必须有基本规范，5.3 节将阐述底层驱动构件的封装规范。

5.2.2　嵌入式硬件构件和软件构件的层次模型

前面提到，在硬件构件中，核心构件为 MCU 的最小系统。通常，MCU 内部包含 GPIO

---

① NATO Communications and Information Systems Agency. NATO Standard for Development of Reusable Software Components[S], 1991.

② 底层驱动构件若不使用 C 语言编程，相应组织形式有变化，但实质不变。

(即通用IO)口和一些内置功能模块,可将通用I/O口的驱动程序封装为GPIO驱动构件,各内置功能模块的驱动程序封装为功能构件,如芯片内含模块的功能构件有串行通信构件、Flash构件、定时器构件等。

在硬件构件层中,相对于核心构件而言,中间构件和终端构件是核心构件的"外设"。由这些"外设"的驱动程序封装而成的软件构件称为底层外设构件。注意,并不是所有的中间构件和终端构件都可以作为编程对象。例如,键盘、LED、LCD等硬件构件与编程有关,而电平转换硬件构件就与编程无关,因而不存在相应的底层驱动程序,当然也就没有相应的软件构件。嵌入式硬件构件与软件构件的层次模型如图5-5所示。

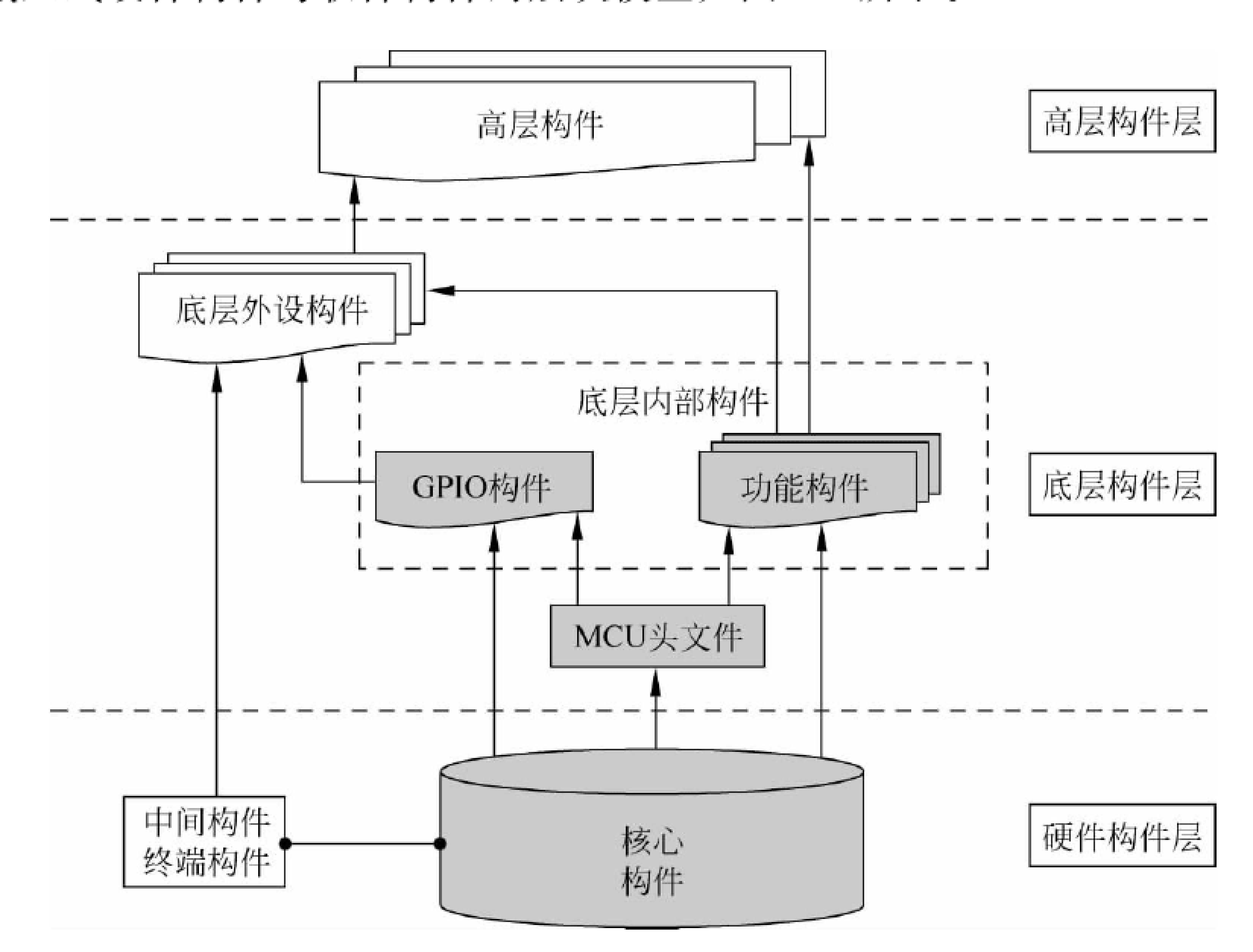


图5-5　嵌入式硬件构件与软件构件的层次模型

由图5-5可看出,底层外设构件可以调用底层内部构件,如LCD构件可以调用GPIO驱动构件、PCF8563构件(时钟构件)可以调用I2C构件等。而高层构件可以调用底层外设构件和底层内部构件中的功能构件,而不能直接调用GPIO驱动构件。另外,考虑到几乎所有的底层内部构件都涉及MCU各种寄存器的使用,因此将MCU的所有寄存器定义组织在一起,形成MCU头文件,以便其他构件头文件中包含该头文件。

5.3　底层驱动构件的封装规范

驱动程序的开发在嵌入式系统的开发中具有举足轻重的地位。驱动程序的好坏直接关系着整个嵌入式系统的稳定性和可靠性。然而,开发出完备、稳定的底层驱动构件并非易事。为了提高底层驱动构件的可移植性和可复用性,特制定本规范。

5.3.1　构件设计的基本思想与基本原则

1. 构件设计的基本思想

底层构件是与硬件直接打交道的软件，它被组织成具有一定独立性的功能模块，由头文件(.h)和源程序文件(.c)两部分组成。构件的头文件名和源程序文件名一致，且为构件名。

构件的头文件中，主要包含必要的引用文件、描述构件功能特性的宏定义语句以及声明对外接口函数。良好的构件头文件应该成为构件使用说明，不需要使用者查看源程序。

构件的源程序文件中包含构件的头文件、内部函数的声明、对外接口函数的实现。

将构件分为头文件与源程序文件两个独立的部分，意义在于，头文件中包含对构件的使用信息的完整描述，为用户使用构件提供充分必要的说明，构件提供服务的实现细节被封装在源程序文件中；调用者通过构件对外接口获取服务，而不必关心服务函数的具体实现细节。这就是构件设计的基本内容。

在设计底层构件时，最关键的工作是要对构件的共性和个性进行分析，设计出合理的、必要的对外接口函数及其形参。**尽量做到：当一个底层构件应用到不同系统中时，仅需修改构件的头文件，对于构件的源程序文件则不必修改或改动很小。**

2. 构件设计的基本原则

在嵌入式软件领域中，由于软件与硬件紧密联系的特性，使得与硬件紧密相连的底层驱动构件的生产成为嵌入式软件开发的重要内容之一。良好的底层驱动构件具备如下特性。

(1) 封装性。在内部封装实现细节，采用独立的内部结构以减少对外部环境的依赖。调用者只通过构件接口获得相应功能，内部实现的调整将不会影响构件调用者的使用。

(2) 描述性。构件必须提供规范的函数名称、清晰的接口信息、参数含义与范围、必要的注意事项等描述，为调用者提供统一、规范的使用信息。

(3) 可移植性。底层构件的可移植性是指同样功能的构件，如何做到不改动或少改动，而方便地移植到同系列及不同系列芯片内，减少重复劳动。

(4) 可复用性。在满足一定使用要求时，构件不经过任何修改就可以直接使用。特别是使用同一芯片开发不同项目，底层驱动构件应该做到复用。可复用性使得高层调用者对构件的使用不因底层实现的变化而有所改变。可复用性提高了嵌入式软件的开发效率、可靠性与可维护性。不同芯片的底层驱动构件复用需在可移植性基础上进行。

为了使构件设计满足**封装性、描述性、可移植性、可复用性**的基本要求，嵌入式底层驱动构件的开发，应遵循**层次化、易用性、鲁棒性及对内存的可靠使用原则。**

1) 层次化原则

层次化设计要求清晰地组织构件之间的关联关系。底层驱动构件与底层硬件打交道，在应用系统中位于最底层。遵循层次化原则设计底层驱动构件需要做到以下几点。

(1) 针对应用场景和服务对象，分层组织构件。设计底层驱动构件的过程中，有一些与处理器相关的、描述了芯片寄存器映射的内容，这些是所有底层驱动构件都需要使用的，将这些内容组织成底层驱动构件的公共内容，作为底层驱动构件的基础。在底层驱动构件的基础上，还可使用高级的扩展构件调用底层驱动构件功能，从而实现更加复杂的服务。

(2) 在构件的层次模型中，**上层构件可以调用下层构件提供的服务，同一层次的构件不存在相互依赖关系，不能相互调用。**例如，Flash 模块与 UART 模块是平级模块，不能在编

写 Flash 构件时，调用 UART 驱动构件。即使要通过 UART 驱动构件函数的调用在 PC 屏幕上显示 Flash 构件测试信息，也不能在 Flash 构件内含有调用 UART 驱动构件函数的语句，应该编写上一层次的程序调用。平级构件是相互不可见的，只有深入理解这一点，并遵守之，才能更好地设计出规范的底层驱动构件。在操作系统下，平级构件不可见特性尤为重要。

2）易用性原则

易用性在于让调用者能够快速理解构件提供服务的功能并进行使用。遵循易用性原则设计底层驱动构件需要做到：**函数名简洁且达意；接口参数清晰，范围明确；使用说明语言精练规范，避免二义性。**此外，在函数的实现方面，避免编写代码量过多。函数的代码量过多会难以理解与维护，并且容易出错。若一个函数的功能比较复杂，可将其"化整为零"，通过编写多个规模较小功能单一的子函数，再进行组合，实现最终的功能。

3）鲁棒性原则

鲁棒性在于为调用者提供安全的服务，避免在程序运行过程中出现异常状况。遵循鲁棒性原则设计底层驱动构件需要做到：**在明确函数输入输出的取值范围、提供清晰接口描述的同时，在函数实现的内部要有对输入参数的检测，对超出合法范围的输入参数进行必要的处理；**使用分支判断时，确保对分支条件判断的完整性，对默认分支进行处理。例如，对 if 结构中的"else"分支和 switch 结构中的"default"安排合理的处理程序。**同时，不能忽视编译警告错误。**

4）内存可靠使用原则

对内存的可靠使用是保证系统安全、稳定运行的一个重要的考虑因素。遵循内存可靠使用原则设计底层驱动构件需要做到以下几点。

(1) 优先使用静态分配内存。相比于人工参与的动态分配内存，静态分配内存由编译器维护，更为可靠。

(2) 谨慎地使用变量。可以直接读写硬件寄存器时，不使用变量替代。避免使用变量暂存简单计算所产生的中间结果。使用变量暂存数据将会影响到数据的时效性。

(3) 检测空指针。定义指针变量时必须初始化，防止产生"野指针"。

(4) 检测缓冲区溢出，并为内存中的缓冲区预留不小于20%的冗余。使用缓冲区时，对填充数据长度进行检测，不允许向缓冲区中填充超出容量的数据。

(5) 对内存的使用情况进行评估。

5.3.2　编码风格基本规范

良好的编码风格能够提高程序代码的可读性和可维护性，而使用统一的编码风格在团队合作编写一系列程序代码时无疑能够提高集体的工作效率。本节给出了编码风格的基本规范，主要涉及文件、函数、变量、宏及结构体类型的命名规范；涉及空格与空行、缩进、断行等的排版规范；涉及文件头、函数头、行及边等的注释规范。

1. 文件、函数、变量、宏及结构体类型的命名规范

命名的基本原则如下。

(1) 命名清晰明了，有明确含义，使用完整单词或约定俗成的缩写。通常，较短的单词可通过去掉元音字母形成缩写；较长的单词可取单词的前几个字母形成缩写，即"见名知

意”。命名中若使用特殊约定或缩写，要有注释说明。

（2）命名风格要自始至终保持一致。

（3）为了代码复用，命名中应避免使用与具体项目相关的前缀。

（4）为了便于管理，对程序实体的命名要体现出所属构件的名称。

（5）使用英语命名。

（6）除宏命名外，名称字符串全部小写，以下画线“_”作为单词的分隔符。首尾字母不用“_”。

针对嵌入式底层驱动构件的设计需要，对文件、函数、变量、宏及数据结构类型的命令特别进行说明。

1）文件的命名

底层驱动构件在具体设计时分为两个文件，其中头文件命名为“<构件名>. h”，源文件命名为“<构件名>. c”，其中，<构件名>表示具体的硬件模块的名称。例如，GPIO 驱动构件对应的两个文件为“gpio. h”和“gpio. c”。

2）函数的命名

底层驱动构件的函数从属于驱动构件，驱动函数的命名除要体现函数功能外，还需要使用命名前缀和后缀标识其所属的构件及不同的实现方式。**函数名前缀：**底层驱动构件中定义的所有函数均使用“<构件名>_”前缀表示其所属的驱动构件模块。例如，GPIO 驱动构件提供的服务接口函数命名为：gpio_init（初始化）、gpio_set（设定引脚状态）、gpio_get（获取引脚状态）等。**函数名后缀：**对同一服务的不同方式的实现，使用后缀加以区分。这样做的好处是：当使用底层构件组装软件系统时，避免构件之间出现同名现象。同时，名称要使人有“顾名思义”的效果。

3）函数形参变量与函数内局部变量的命名

对嵌入式底层驱动构件进行编码的过程中，需要考虑对底层驱动函数形参变量及驱动函数内部局部变量的命名。**函数形参变量：**函数形参变量名是使用函数时理解形参的最直观印象，表示传参的功能说明。特别地，若传入底层驱动函数接口的参数是指针类型，则在命名时应使用“_ptr”后缀加以标识。**局部变量：**局部变量的命名与函数形参变量类似。但函数形参变量名一般不取单个字符（如 i、j、k）进行命名，而 i、j、k 作局部循环变量是允许的。这是因为，变量，尤其是局部变量，如果用单个字符表示，很容易写错（如 i 写成 j），而编译时又检查不出来，有可能为了这个小小的错误而花费大量的查错时间。

4）宏常量及宏函数的命名

宏常量及宏函数的命名全部使用大写字符，使用下画线“_”为分隔符。例如，在构件公共要素中定义的开关中断的宏如下：

```
#define ENABLE_INTERRUPTS asm(" CPSIE i")        //开总中断
#define DISABLE_INTERRUPTS asm(" CPSID i")       //关总中断
```

5）结构体类型的命名、类型定义与变量声明

（1）结构体类型名称使用小写字母命名（< defined_struct_name >），定义结构体类型变量时，全部使用大写字母命名（< DEFINED_STRUCT_NAME >）。

（2）对结构体内部字段全部使用大写字母命名（< ELEM_NAME >）。

(3) 定义类型时,同时声明一个结构体变量和结构体指针变量。

模板如下:

```
typedef struct <defined_struct_name>
{
<elem_type_1><ELEM_NAME_1>;        //对字段 1 含义的说明
<elem_type_2><ELEM_NAME_2>;        //对字段 2 含义的说明
...
} <DEFINED_STRUCT_NAME>, * <DEFINED_STRUCT_NAME_PTR>;
```

例如,当要定义一个描述 UART 设备初始化参数结构体类型时,可有如下定义:

```
typedef struct uart_init
{
    uint_8   DEV_ID:            //串口设备号
    uint_32  BAUD_RATE:         //串口通信波特率
} UART_INIT_STRUCT, * UART_INIT_PTR;
```

这样,"uart_init"就是一种结构体类型,而 UART_INIT_STRUCT 是一个 uart_init 类型变量,UART_INIT_PTR 是 uart_init 类型指针变量。

2. 排版

对程序进行排版是指,通过插入空格与空行,使用缩进、断行等手段,调整代码的书面版式,使代码整体美观、清晰,从而提高代码的可读性。

1) 空行与空格

关于空行:相对独立的程序块之间须加空行。

关于空格:在两个以上的关键字、变量、常量进行对等操作时,它们之间的操作符之前、之后或者前后要加空格,必要时加两个空格;进行非对等操作时,如果是关系密切的立即操作符(如->),其后不应加空格。采用这种松散方式编写代码的目的是使代码更加清晰。例如,只在逗号、分号后面加空格;在比较操作符,赋值操作符"=""+=",算术操作符"+""%",逻辑操作符"&&",位域操作符"<<""^"等双目操作符的前后加空格;在"!""~""++""--""&"(地址运算符)等单目操作符前后不加空格;在"->""."前后不加空格;在 if、for、while、switch 等与后面括号间加空格,使关键字更为突出、明显。

2) 缩进

使用空格缩进,建议不使用 Tab 键,这样代码复制打印时不会造成错乱。代码的每一级均往右缩进 4 个空格的位置。函数或过程的开始、结构的定义及循环、判断等语句中的代码都要采用缩进风格,case 语句下的情况处理语句也要遵从语句缩进要求。

3) 断行

(1) **较长的语句(>78 字符)要分成多行书写,**长表达式要在低优先级操作符处划分新行,操作符放在新行之首,划分出的新行要进行适当的缩进,使排版整齐,语句可读。

(2) 循环、判断等语句中若有较长的表达式或语句,则要进行适应的划分,长表达式要在低优先级操作符处划分新行,操作符放在新行之首。

(3) 若函数或过程中的参数较长,则要进行适当的划分。

(4) 不允许把多个短语句写在一行中，即一行只写一条语句。特殊情况可用，例如“if(x>3)x=3;”可以在一行。

(5) if、for、do、while、case、switch、default 等语句后的程序块分界符(如 C/C++ 语言的大括号“{”和“}”)应各独占一行并且位于同一列，且与以上保留字左对齐。

3. 注释

在程序代码中使用注释，有助于对程序的阅读理解，说明程序在“做什么”，解释代码的目的、功能和采用的方法。编写注释时要注意以下几点。

(1) 一般情况源程序有效注释量在 30%左右。

(2) 注释语言必须准确、易懂、简洁。

(3) 编写和修改代码的同时，处理好相应的注释。

(4) **C 语言中采用“//”注释，不使用段注释“/* */”。保留段注释用于调试，便于注释不用的代码。**

为规范嵌入式底层驱动构件的注释，特别对文件头注释、函数头注释、行注释与边注释进行特别说明。

1) 文件头注释

底层驱动构件的接口头文件和实现源文件的开始位置，使用文件头注释，如：

```
//======================================================================
//文件名称：gpio.h
//功能概要：GPIO 底层驱动构件头文件
//版权所有：苏州大学嵌入式中心(sumcu.suda.edu.cn)
//版本更新：2016-03-12 V1.0
//======================================================================
```

2) 函数头注释

在驱动函数的接口声明和函数实现前，使用函数头注释详细说明驱动函数提供的服务。在构件的头文件中必须添加完整的函数头注释，为构件使用者提供充分的使用信息。构件的源文件对用户是透明的，因此，在必要时可适当简化函数头注释的内容。例如：

```
//======================================================================
//函数名称：gpio_init
//函数返回：无
//参数说明：port_pin：(端口号)|(引脚号)(例：PTB_NUM|5 表示为 B 口 5 号脚)
//          dir：引脚方向(0=输入,1=输出,可用引脚方向宏定义)
//          state：端口引脚初始状态(0=低电平,1=高电平)
//功能概要：初始化指定端口引脚作为 GPIO 引脚功能，并定义为输入或输出，若是输出，
//          还指定初始状态是低电平或高电平
//======================================================================
```

3) 整行注释与边注释

整行注释文字，主要是对至下一个整行注释之前的代码进行功能概括与说明。边注释位于一行程序的尾端，对本语句或至下一边注释之间的语句进行功能概括与说明。此外，分支语句(条件分支、循环语句等)须在结束的“}”右方进行边注释，表明该程序块结束的标记

“end_…”，尤其在多重嵌套时。对于有特别含义的变量、常量，如果其命名不是充分自注释的，在声明时都必须加以注释，说明其含义。变量、常量、宏的注释应放在其上方相邻位置（行注释）或右方（边注释）。

5.3.3　公共要素文件

为某一款芯片编写驱动构件时，不同的构件存在公共使用的内容，将这些内容以构件的形式组织起来，称为构件公共要素。构件公共要素在底层驱动构件的体系中有着特殊的地位，为设备底层驱动构件的编写提供最基本的支持。在不同的应用环境间移植驱动构件时，都应根据软硬件的基本情况在构件公共要素文件中进行相关的配置，满足所有底层驱动构件正常工作时所需的基本和公共需求。所有底层驱动构件都包含对构件公共要素的引用。构件公共要素文件放在工程文件夹的“\Common”文件夹下，名为common.h。本节以common.h文件内容为主线，介绍构件公共要素提供的服务。

1. 芯片寄存器映射文件

每个底层驱动构件都是以硬件模块的特殊功能寄存器为操作对象，因此，在common.h文件中包含描述芯片寄存器映射的头文件，当底层驱动构件引用common.h文件时，即可使用片内寄存器映射文件中定义访问各自相关的特殊功能寄存器。

除包含芯片片内寄存器映像文件，还需要将内核及芯片相关文件引用到公共要素中。这些文件的功能简介见4.5节内容。一般地，开关总中断是嵌入式编程中常用的功能，当运行某些程序不希望被外部事件打断时，就可以暂时关闭中断系统，为程序运行提供一个“安静”的运行环境。开关总中断是嵌入式编程中常用的功能，C语言的编译器无法为具体的芯片生成开关总中断的语句。

```
#define ENABLE_INTERRUPTS __enable_irq            //开总中断
#define DISABLE_INTERRUPTS __disable_irq          //关总中断
```

在core_cmFunc.h文件中可以看出，函数__enable_irq和__disable_irq分别是内嵌汇编的方式定义开关中断的语句，所以开关总中断的宏定义语句等同于以下语句：

```
#define ENABLE_INTERRUPTS asm(" CPSIE i")         //开总中断
#define DISABLE_INTERRUPTS asm(" CPSID i")        //关总中断
```

2. 一位操作的宏函数

将编程时经常用到的对寄存器的某一位进行操作，即对寄存器的置位、清位及获得寄存器某一位状态的操作，定义成宏函数。设置寄存器某一位为1，称为置位；设置寄存器某一位为0，称为清位。这在底层驱动编程时经常用到。置位与清位的基本原则是：当对寄存器的某一位进行置位或清位操作时，不能干扰该寄存器的其他位，否则，可能会出现意想不到的错误。

综合利用“<<”“>>”“|”“&”“~”等位运算符，可以实现置位与清位，且不影响其他位的功能。下面以8位寄存器为例进行说明，其方法适用于各种位数的寄存器。设R为8位寄存器，下面说明将R的某一位置位与清位，而不干预其他位的编程方法。

(1) 置位。要将R的第3位置1，其他位不变，可以这样做：R |= (1 << 3)，其中“1 <<

3”的结果是“0b00001000”,R |= (1 << 3)也就是R=R|0b00001000,任何数和0相或不变,任何数和1相或为1,这样达到对R的第3位置1,但不影响其他位的目的。

(2) 清位。要将R的第2位清0,其他位不变,可以这样做:R &= ~(1 << 2),其中“~(1 << 2)”的结果是“0b11111011”,R&=~(1 << 2)也就是R=R&0b11111011,任何数和1相与不变,任何数和0相与为0,这样达到对R的第2位清0,但不影响其他位的目的。

(3) 获得某一位的状态。(R >> 4) & 1,是获得R第4位的状态,“R >> 4”是将R右移4位,将R的第4位移至第0位,即最后一位,再和1相与,也就是和0b00000001相与,保留R最后一位的值,以此得到第4位的状态值。

为了方便使用,把这种方法改为带参数的“宏函数”,并且简明定义,放在公共头文件(common.h)中。使用该“宏”的文件,可以包含“common.h”文件。

```
#define BSET(bit,Register) ((Register) |= (1<<(bit)))      //置Register的第bit位为1
#define BCLR(bit,Register) ((Register) &= ~(1<<(bit)))     //清Register的第bit位
#define BGET(bit,Register) (((Register) >> (bit)) & 1)     //取Register的第bit位状态
```

这样就可以使用BSET、BCLR、BGET这些容易理解与记忆的标识,进行寄存器的置位、清位及获得寄存器某一位状态的操作。

3. 重定义基本数据类型

嵌入式程序设计与一般的程序设计有所不同,在嵌入式程序中打交道的大多数都是底层硬件的存储单元或是寄存器,所以在编写程序代码时,使用的基本数据类型多以8位、16位、32位数据长度为单位。不同的编译器为基本整型数据类型分配的位数存在不同,但在编写嵌入式程序时要明确使用变量的字长,因此,需根据具体编译器重新定义嵌入式基本数据类型。重新定义后,不仅书写方便,也有利于软件的移植。例如:

```
//重定义基本数据类型(类型别名宏定义)
typedef unsigned char       uint_8;     //无符号8位数,字节
typedef unsigned short int  uint_16;    //无符号16位数,字
typedef unsigned long int   uint_32;    //无符号32位数,长字
typedef char                int_8;      //有符号8位数
typedef short int           int_16;     //有符号16位数
typedef int                 int_32;     //有符号32位数
//不优化类型
typedef volatile uint_8     vuint_8;    //不优化无符号8位数,字节
typedef volatile uint_16    vuint_16;   //不优化无符号16位数,字
typedef volatile uint_32    vuint_32;   //不优化无符号32位数,长字
typedef volatile int_8      vint_8;     //不优化有符号8位数
typedef volatile int_16     vint_16;    //不优化有符号16位数
typedef volatile int_32     vint_32;    //不优化有符号32位数
```

通常有一些数据类型不能进行优化处理。在此,对不优化数据类型的定义做特别说明。不优化数据类型的修饰关键字是**volatile**。它用于通知编译器,对其后面所定义的变量不能随意进行优化,因此,编译器会安排该变量使用系统存储区的具体地址单元,编译后的程序每次需要存储或读取该变量时,都会直接访问该变量的地址。若没有volatile关键字,则编译器可能会暂时使用CPU寄存器来存储,以优化存储和读取,这样,CPU寄存器和变量地

址的内容很可能会出现不一致现象。对 MCU 的映像寄存器的操作就不能优化,否则,对 I/O 口的写入可能被"优化"写入到 CPU 内部寄存器中,就会乱套。常用的 volatile 变量使用场合有设备的硬件寄存器、中断服务例程中访问到的非自动变量、操作系统环境下多线程应用中被几个任务共享的变量。

5.3.4 头文件的设计规范

头文件描述了构件的接口,用户通过头文件获取构件服务。在本节中,对底层驱动构件头文件的内容的编写加以规范,从程序编码结构、包含文件的处理、宏定义及设计服务接口等方面进行说明。

1. 编码框架

编写每个构件的头文件时,应使用"#ifndef… #define … #endif"编码结构,防止对头文件的重复包含。例如,若定义 GPIO 驱动构件,在其头文件 gpio. h 中,应有:

```
#ifndef _GPIO_H
#define _GPIO_H
…                          //文件内容
#endif
```

2. 包含文件

包含文件命令为#include,包含文件的语句统一安排在构件的头文件中,而在相应构件的源文件中仅包含本构件的头文件。将包含文件的语句统一置于构件的头文件中,使文件间的引用关系能够更加清晰地呈现。

3. 使用宏定义

宏定义命令为#define,使用宏定义可以替换代码内容,替换内容可以是常数、字符串,甚至还可以是带参数的函数。利用宏定义的替换特性,当需要变更程序的宏常量或宏函数时,只需一次性修改宏定义的内容,程序中每个出现宏常量或宏函数的地方均会自动更新。

(1) 使用宏定义表示构件中的常量,为常量值提供有意义的别名。

比如,在 Light 构件(指示灯构件)中使用 GPIO 驱动构件,灯的亮暗状态与对应 GPIO 引脚高低电平的对应关系需根据外接电路而定,此时,将表示灯状态的电平信号值用宏常量的方式定义。当使用的外部电路发生变化时,对应地,仅需在 Light 构件中对表示灯的亮暗状态宏常量定义做适当变更,就可实现 Light 构件在新应用环境上的移植。

```
#define LIGHT_ON     0          //灯亮
#define LIGHT_OFF    1          //灯暗
```

(2) 使用宏函数实现构件对外部请求服务的接口映射。

在设计构件时,有时会需要应用环境为构件的基本活动提供服务。此时,采用宏函数表示构件对外部请求服务的接口,在构件中不关心请求服务的实现方式,这就为构件在不同应用环境下的移植提供了较强的灵活性。

4. 声明对外接口函数,包含对外接口函数的使用说明

底层驱动构件通过外接口函数为调用者提供简明而完备的服务,对外接口函数的声明

及使用说明(即函数的头注释)存于头文件中。外接口函数的设计规范见5.3.5节。

5.3.5　源程序文件的设计规范

编写底层驱动构件实现源文件基本要求,是实现构件通过服务接口对外提供全部服务的功能。为确保构件工作的独立性,实现构件高内聚、低耦合的设计要求,将构件的实现内容封装在源文件内部。对于底层驱动构件的调用者而言,通过服务接口获取服务,而不需要了解驱动构件提供服务的具体运行细节。因此,功能实现和封装是编写底层驱动构件实现源文件的主要考虑内容。

1. 源程序文件中的#include

底层驱动构件的源文件(.c)中,只允许一处使用#include包含自身头文件。需要包含的内容需在自身构件的头文件中包含,以便有统一、清晰的程序结构。

2. 合理设计与实现对外接口函数与内部函数

驱动构件的源程序文件中的函数包含对外接口函数与内部函数。对外接口函数,供上层应用程序调用,其头注释需完整表述函数名、函数功能、入口参数、函数返回值、使用说明、函数适用范围等信息,以增强程序的可读性。在构件中的封装比较复杂功能的函数时,代码量不宜过长,此时,就应当将其中功能相对独立的部分封装成子函数。这些子函数仅在构件内部使用,不提供对外服务,因此被称为"内部函数"。为将内部函数的访问范围限制在构件的源文件内部,在创建内部函数时,应使用static关键字作为修饰符。内部函数的声明放在所有对外接口函数程序的上部,代码实现放在对外接口函数程序的后部。

一般地,实现底层驱动构件的功能,需要同芯片片内模块的特殊功能寄存器打交道,通过对相应寄存器的配置实现对设备的驱动。某些配置过程对配置的先后顺序和时序有特殊要求,在编写驱动程序时要特别注意。

对外接口函数实现完成后,复制其头注释于头文件中,作为构件的使用说明。参考样例见网上教学资源的GPIO构件及Light构件(各样例工程下均有)。

3. 不使用全局变量

全局变量的作用范围可以扩大到整个应用程序,其中存放的内容在应用程序的任何一处都可以随意修改,一般可用于在不同程序单元间传递数据。但是,若在底层驱动构件中使用全局变量,其他程序即使不通过构件提供的接口也可以访问到构件内部,这无疑对构件的正常工作带来隐患。从软件工程理论中对封装特性的要求上看,也不利于构件设计高内聚、低耦合的要求。因此,在编写驱动构件程序时,**严格禁止使用全局变量**。用户与构件交互只能通过服务接口进行,即所有的数据传递都要通过函数的形参来接收,而不是使用全局变量。

5.4　硬件构件及底层软件构件的重用与移植方法

重用是指在一个系统中,同一构件可被重复使用多次。移植是指将一个系统中使用到的构件应用到另外一个系统中。

1. 硬件构件的重用与移植

对于以单 MCU 为核心的嵌入式应用系统而言,当用硬件构件"组装"硬件系统时,核心构件(即最小系统)有且只有一个,而中间构件和终端构件可有多个,并且相同类型的构件可出现多次。下面以终端构件 LCD 为例,介绍硬件构件的移植方法。

在应用系统 A 中,若 LCD 的数据线(LCD-D0～LCD-D7)与芯片的通用 IO 口的 B 口相连,C 口作为 LCD 的控制信号传送口,其中,LCD 寄存器选择信号 LCD-RS 与 C 口第 0 引脚连接,读写信号 LCD-RW 与 C 口第 1 引脚连接,使能信号 LCD_E 与 C 口第 2 引脚连接,则 LCD 硬件构件实例如图 5-6(a)所示。虚线框左边的文字(如 PTC0、PTC1 等)为接口网标,虚线框右边的文字(如 LCD-RS、LCD-RW 等)为接口注释。

在应用系统 B 中,若 LCD 的数据线(LCD-D0～LCD-D7)与 MCF52233(32 位 MCU)芯片的通用 IO 口的 AN 口相连,TA 口的第 0、1、2 引脚分别作为寄存器选择信号 LCD-RS、读写信号 LCD-RW、使能信号 LCD_E,则 LCD 硬件构件实例如图 5-6(b)所示。

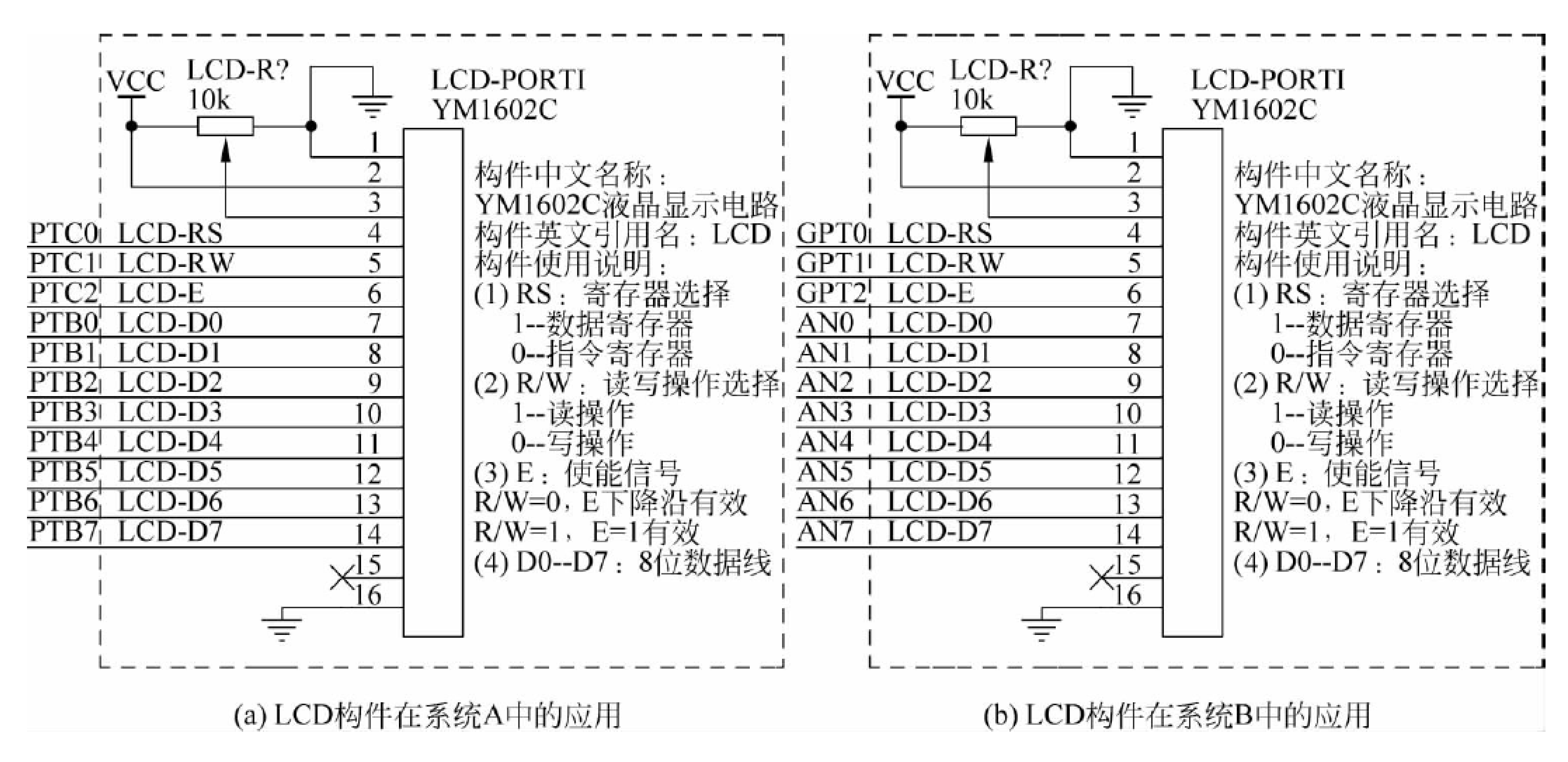


图 5-6　LCD 构件在实际系统中的应用

2. 底层构件的移植

当一个已设计好的底层构件移植到另外一个嵌入式系统中时,其头文件和程序文件是否需要改动呢?这要视具体情况而定。例如,系统的核心构件发生改变(即 MCU 型号改变)时,底层内部构件头文件和某些对外接口函数也要随之改变,例如模块初始化函数。

而对于外接硬件构件,希望不改动程序文件,而只改动头文件,那么,头文件就必须充分设计。以 LCD 构件为例,与图 5-6(a)相对应的底层构件头文件 lcd.h 可如下编写。

```
//==========================================
// 文件名称: lcd.h
// 功能概要: lcd 构件头文件
// 版权所有: 苏州大学嵌入式中心(sumcu.suda.edu.cn)
// 版本更新: 2013-03-17,V1.02016-03-12,V3.0(WYH)
//==========================================

#ifndef LCD_H
#define LCD_H
```

```
#include "common.h"
#include "gpio.h"

#define LCDRS     PTC_NUM|(0)      //LCD 寄存器选择信号
#define LCDRW     PTC_NUM|(1)      //LCD 读写信号
#define LCDE      PTC_NUM|(2)      //LCD 读写信号
//LCD 数据引脚
#define LCD_D7    PTB_NUM|(7)
#define LCD_D6    PTB_NUM|(6)
#define LCD_D5    PTB_NUM|(5)
#define LCD_D4    PTB_NUM|(4)
#define LCD_D3    PTB_NUM|(3)
#define LCD_D2    PTB_NUM|(2)
#define LCD_D1    PTB_NUM|(1)
#define LCD_D0    PTB_NUM|(0)
//==========================================
//函数名称：LCDInit
//函数返回：无
//参数说明：无
//功能概要：LCD 初始化
//==========================================
void LCDInit();

//==========================================
//函数名称：LCDShow
//函数返回：无
//参数说明：data[32]：需要显示的数组
//功能概要：LCD 显示数组的内容
//==========================================
void LCDShow(uint_8 data[32]);

#endif
```

当 LCD 硬件构件发生如图 5-6(b)所示的移植时，显示数据传送口和控制信号传送口发生了改变，只需修改头文件，而不需修改 lcd.c 文件。

必须申明的是，本书给出构件化设计方法的目的是，在进行软硬件移植时，设计人员所做的改动要尽量小，而不是不做任何改动。希望改动尽可能在头文件中进行，而不希望改动程序文件。

小　　结

本章属于方法论内容，与具体芯片无关，主要阐述嵌入式硬件构件及底层驱动构件的基本规范。

(1) 机械、建筑等传统产业的运作模式是先生产符合标准的构件(零部件)，然后将标准构件按照规则组装成实际产品。构件是核心和基础，复用是必需的手段。嵌入式软硬件设

计也借助于这个概念。嵌入式硬件构件是指将一个或多个硬件功能模块、支撑电路及其功能描述封装成一个可重用的硬件实体,并提供一系列规范的输入/输出接口。嵌入式硬件构件根据接口之间的生产消费关系,接口可分为供给接口和需求接口两类。根据所拥有接口类型的不同,硬件构件分为核心构件、中间构件和终端构件三种类型。核心构件只有供给接口,没有需求接口,它只为其他硬件构件提供服务,而不接受服务。中间构件既有需求接口又有供给接口,它不仅能够接受其他构件提供的服务,而且也能够为其他构件提供服务。终端构件只有需求接口,它只接受其他构件提供的服务。设计核心构件时,需考虑的问题是:"核心构件能为其他构件提供哪些信号?"设计中间构件时,需考虑的问题是:"中间构件需要接收哪些信号,以及提供哪些信号?"设计终端构件时,需考虑的问题是:"终端构件需要什么信号才能工作?"

(2) 嵌入式底层驱动构件是直接面向硬件操作的程序代码及使用说明。规范的底层驱动构件由头文件(.h)及源程序文件(.c)文件构成。头文件(.h)是底层驱动构件简明且完备的使用说明,即在不查看源程序文件情况下,就能够完全使用该构件进行上一层程序的开发,这也是设计底层驱动构件最值得遵循的原则。

(3) 在设计实现驱动构件的源程序文件时,需要合理设计外接口函数与内部函数。外接口函数,供上层应用程序调用,其头注释需完整表述函数名、函数功能、入口参数、函数返回值、使用说明、函数适用范围等信息,以增强程序的可读性。在具体代码实现时,严格禁止使用全局变量。

(4) 在嵌入式硬件原理图设计中,要充分利用嵌入式硬件进行复用设计;在嵌入式软件编程时,涉及与硬件直接打交道时,应尽可能复用底层驱动构件。若无可复用的底层驱动构件,应该按照基本规范设计驱动构件,然后再进行应用程序开发。

习　题

1. 简述嵌入式硬件构件概念及嵌入式硬件构件分类。
2. 简述核心构件、中间构件和终端构件的含义及设计规则。
3. 阐述嵌入式底层驱动构件的基本内涵。
4. 在设计嵌入式底层驱动构件时,其对外接口函数设计的基本原则有哪些?
5. 举例说明在什么情况下使用宏定义。
6. 举例说明底层构件的移植方法。
7. 利用C语言,自行设计一个底层驱动构件,并进行调试。
8. 利用一种汇编语言,设计一个底层驱动构件,并进行调试,同时与C语言设计的底层驱动构件进行简明比较。

第 6 章　串行通信模块及第一个中断程序结构

本章导读：本章阐述 KL25/26 的串行通信模块构件化编程。主要内容有：①异步串行通信(UART)的通用基础知识，着重给出异步串行通信的格式与波特率概念，简要介绍 RS232 总线标准，给出串行通信编程模型；②KL25/26 芯片 UART 驱动构件及使用方法，给出测试实例，这是从实际应用角度阐述异步串行通信；③ARM Cortex-M0+中断机制及 KL25/26 中断编程步骤，这是本书给出完整中断编程实例，目的是阐述嵌入式系统的中断处理基本方法；④UART 驱动构件的设计方法，主要是 UART 驱动构件设计需要的相关寄存器，并给出 UART 驱动构件的主要实现代码，这个部分可根据实际教学情况选用。

本章参考资料：6.2.2 节(KL25/26 芯片 UART 引脚)参考自《KL 参考手册》的第 10 章；6.3.2 节有关 M0+的中断机制总结自《M0+参考手册》第 5 章以及《KL 参考手册》第 3.3 节；6.4.1 节(UART 模块编程结构)参考自《KL 参考手册》的第 39、40 章。

6.1　异步串行通信的通用基础知识

串行通信接口，简称“串口”、UART 或 SCI。在 USB 未普及之前，串口是 PC 必备的通信接口之一。作为设备间简便的通信方式，在相当长的时间内，串口还不会消失，在市场上也可很容易的购买到各种电平到 USB 的串口转接器，以便与没有串口但具有多个 USB 口的笔记本或 PC 连接。MCU 中的串口通信，在硬件上一般只需要三根线，分别称为发送线(TxD)、接收线(RxD)和地线(GND)；通信方式上，属于单字节通信，是嵌入式开发中重要的打桩调试手段。实现串口功能的模块在一部分 MCU 中被称为通用异步收发器(Universal Asynchronous Receiver-Transmitters，UART)，在另一些 MCU 中被称为串行通信接口(Serial Communication Interface，SCI)。

本节简要概述 UART 的基本概念与硬件连接方法，为学习 MCU 的 UART 编程做准备。

6.1.1　串行通信的基本概念

“位”(bit)是单个二进制数字的简称，是可以拥有两种状态的最小二进制值，分别用“0”和“1”表示。在计算机中，通常一个信息单位用 8 位二进制表示，称为一个“字节”(Byte)。串行通信的特点是：数据以字节为单位，按位的顺序(例如最高位优先)从一条传输线上发送出去。这里至少涉及以下几个问题：第一，每个字节之间是如何区分开的？第二，发送一位的持续时间是多少？第三，怎样知道传输是正确的？第四，可以传输多远？这些问题属于串行通信的基本概念。串行通信分为异步通信与同步通信两种方式，本节主要给出异步串行通信的一些常用概念。正确理解这些概念，对串行通信编程是有益的。**主要掌握异步串行通信的格式与波特率，至于奇偶校验与串行通信的传输方式术语了解即可。**

1. 异步串行通信的格式

在MCU的英文芯片手册上，通常说的异步串行通信采用的是NRZ数据格式，英文全称是："standard non-return-zero mark/space data format"，可以译为："标准不归零传号/空号数据格式"。这是一个通信术语，"不归零"的最初含义是：用负电平表示一种二进制值，正电平表示另一种二进制值，不使用零电平。"mark/space"即"传号/空号"分别表示两种状态的物理名称，逻辑名称记为"1/0"。对学习嵌入式应用的读者而言，只要理解这种格式只有"1""0"两种逻辑值就可以了。图6-1给出了8位数据、无校验情况的传送格式。



图6-1　串行通信数据格式

这种格式的空闲状态为"1"，发送器通过发送一个"0"表示一个字节传输的开始，随后是数据位(在MCU中一般是8位或9位，可以包含校验位)。最后，发送器发送1位或2位的停止位，表示一个字节传送结束。若继续发送下一字节，则重新发送开始位(这就是异步的含义了)，开始一个新的字节传送。若不发送新的字节，则维持"1"的状态，使发送数据线处于空闲。从开始位到停止位结束的时间间隔称为一字节帧(Byte Frame)。所以，也称这种格式为字节帧格式。每发送一个字节，都要发送"开始位"与"停止位"，这是影响异步串行通信传送速度的因素之一。

2. 串行通信的波特率

位长(Bit Length)，也称为位的持续时间(Bit Duration)，其倒数就是单位时间内传送的位数。人们把每秒内传送的位数叫作波特率(Baud Rate)。波特率的单位是：位/秒，记为bps。bps是英文bit per second的缩写，习惯上这个缩写不用大写，而用小写。通常情况下，波特率的单位可以省略。

通常使用的波特率有1200、1800、2400、4800、9600、19 200、38 400、57 600和115 200等。在包含开始位与停止位的情况下，发送一个字节需10位，很容易计算出，在各波特率下，发送1KB所需的时间。显然，这个速度相对于目前许多通信方式而言是慢的，那么，异步串行通信的速度能否提得很高呢？答案是不能。因为随着波特率的提高，位长变小，以至于很容易受到电磁源的干扰，通信就不可靠了。当然，还有通信距离问题，距离小，可以适当提高波特率，但这样毕竟提高的幅度非常有限，达不到大幅度提高的目的。

3. 奇偶校验

在异步串行通信中，如何知道一个字节的传输是否正确？最常见的方法是增加一个位(奇偶校验位)，供错误检测使用。字符奇偶校验检查(Character Parity Checking)称为垂直冗余检查(Vertical Redundancy Checking，VRC)，它是为每个字符增加一个额外位使字符中"1"的个数为奇数或偶数。奇数或偶数依据使用的是"奇校验检查"还是"偶校验检查"而定。当使用"奇校验检查"时，如果字符数据位中"1"的数目是偶数，校验位应为"1"，如果"1"的数目是奇数，校验位应为"0"。当使用"偶校验检查"时，如果字符数据位中"1"的数目是偶数，则校验位应为"0"，如果是奇数则为"1"。这里列举奇偶校验检查的一个实例，看看ASCII字符"R"，其位构成是1010010。由于字符"R"中有三个位为"1"，若使用奇校验检查，则校验位为0；如果使用偶校验检查，则校验位为1。

在传输过程中，若有一位(或奇数个数据位)发生错误，使用奇偶校验检查，可以知道发生传输错误。若有两位(或偶数个数据位)发生错误，使用奇偶校验检查，就不能知道已经发生了传输错误。但是奇偶校验检查方法简单，使用方便，发生一位错误的概率远大于两位的概率，所以"奇偶校验"这种方法还是最为常用的校验方法。几乎所有 MCU 的串行异步通信接口，都提供这种功能。但实际编程中使用较少，原因是单字节校验意义不大。

4. 串行通信传输方式术语

在串行通信中，经常用到"单工""双工""半双工"等术语，它们是串行通信的不同传输方式。下面简要介绍这些术语的基本含义。

(1) 全双工(Full-duplex)：数据传送是双向的，且可以同时接收与发送数据。这种传输方式中，除了地线之外，需要两根数据线，站在任何一端的角度看，一根为发送线，另一根为接收线。一般情况下，MCU 的异步串行通信接口均是全双工的。

(2) 半双工(Half-duplex)：数据传送也是双向的，但是在这种传输方式中，除地线之外，一般只有一根数据线。任何时刻，只能由一方发送数据，另一方接收数据，不能同时收发。

(3) 单工(Simplex)：数据传送是单向的，一端为发送端，另一端为接收端。这种传输方式中，除了地线之外，只要一根数据线就可以了。有线广播就是单工的。

6.1.2 RS232 总线标准

现在回答"可以传输多远"这个问题。MCU 引脚输入/输出一般使用 TTL(Transistor Transistor Logic)电平，即晶体管-晶体管逻辑电平。而 TTL 电平的"1"和"0"的特征电压分别为 2.4V 和 0.4V(目前使用 3V 供电的 MCU 中，该特征值有所变动)，即大于 2.4V 则识别为"1"，小于 0.4V 则识别为"0"。它适用于板内数据传输。若用 TTL 电平将数据传输到 5m 之外，那么可靠性就很值得考究了。为使信号传输得更远，美国电子工业协会(Electronic Industry Association，EIA)制定了串行物理接口标准 RS232C，以下简称 RS232。RS232 采用负逻辑，－15～－3V 为逻辑"1"，＋3～＋15V 为逻辑"0"。RS232 最大的传输距离是 30m，通信速率一般低于 20Kbps。当然，在实际应用中，也有人用降低通信速率的方法，通过 RS232 电平，将数据传送到 300m 之外，这是很少见的，且稳定性很不好。

RS232 总线标准最初是为远程数据通信制定的，但目前主要用于几米到几十米范围内的近距离通信。有专门的书籍介绍这个标准，但对于一般的读者，不需要掌握 RS232 标准的全部内容，只要了解本节介绍的这些基本知识就可以使用 RS232。目前一般的 PC 均带有一两个串行通信接口，人们也称之为 RS232 接口，简称"串口"，它主要用于连接具有同样接口的室内设备。早期的标准串行通信接口是 25 芯插头，这是 RS232 规定的标准连接器(其中，两条地线，4 条数据线，11 条控制线，三条定时信号，其余 5 条线备用或未定义)。

后来，人们发现在计算机的串行通信中，25 芯线中的大部分并不使用，逐渐改为使用 9 芯串行接口。一段时间内，市场上还有 25 芯与 9 芯的转接头，方便了两种不同类型之间的转换。后来，使用 25 芯串行插头极少见到，25 芯与 9 芯转接头也极少有售。因此，目前几乎所有计算机上的串行口都是 9 芯接口。图 6-2 给出了 9 芯串行接口的排列位置，相应引脚含义见表 6-1。

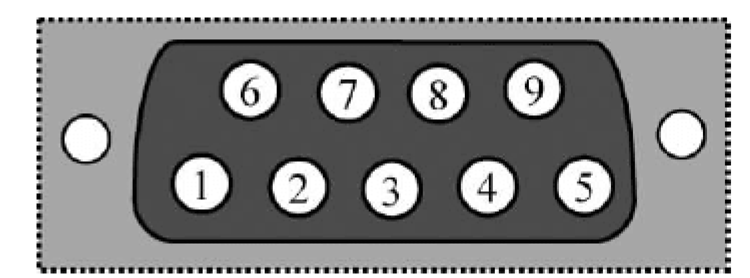


图 6-2　9 芯串行接口排列

表 6-1　计算机中常用的 9 芯串行接口引脚含义表

引脚号	功　能	引脚号	功　能
1	接收线信号检测(载波检测 DCD)	6	数据通信设备准备就绪(DSR)
2	接收数据线(RxD)	7	请求发送(RTS)
3	发送数据线(TxD)	8	允许发送(CTS)
4	数据终端准备就绪(DTR)	9	振铃指示
5	信号地(SG)		

在 RS232 通信中,常常使用精简的 RS232 通信,通信时仅使用三根线:RxD(接收线)、TxD(发送线)和 GND(地线)。其他为进行远程传输时接调制解调器之用,有的也可作为硬件握手信号(如请求发送 RTS 信号与允许发送 CTS 信号),初学时可以忽略这些信号的含义。

此外,为了组网方便,还有一种标准,称为 RS485,RS485 采用差分信号负逻辑,−6～−2V 表示"1",+2～+6V 表示"0"。硬件连接上,采用两线制接线方式,工业应用较多。但由于 PC 默认只带有 RS232 接口,因此,市场上有 RS232-RS485 转接头出售。但这些均是硬件电平信号之间的转换,与 MCU 编程无关。

6.1.3　TTL 电平到 RS232 电平转换电路

在 MCU 中,若用 RS232 总线进行串行通信,则需外接电路实现电平转换。在发送端,需要用驱动电路将 TTL 电平转换成 RS232 电平;在接收端,需要用接收电路将 RS232 电平转换为 TTL 电平。电平转换器不仅可以由晶体管分立元件构成,也可以直接使用集成电路。目前广泛使用 MAX232 芯片较多,图 6-3 给出了 MAX232 的引脚说明,引脚含义简要说明如下:Vcc(16 脚)正电源端,一般接+5V;GND(15 脚)地端;VS+(2 脚):VS+=2Vcc−1.5V=8.5V;VS−(6 脚):VS−=−2Vcc−1.5V=−11.5V;C2+、C2−(4、5 脚):一般接 1μF 的电解电容;C1+、C1−(1、3 脚):一般接 1μF 的电解电容。输入输出引脚分为两组,基本含义见表 6-2。在实际使用时,若只需要一路串行通信接口,可以使用其中的任何一组。

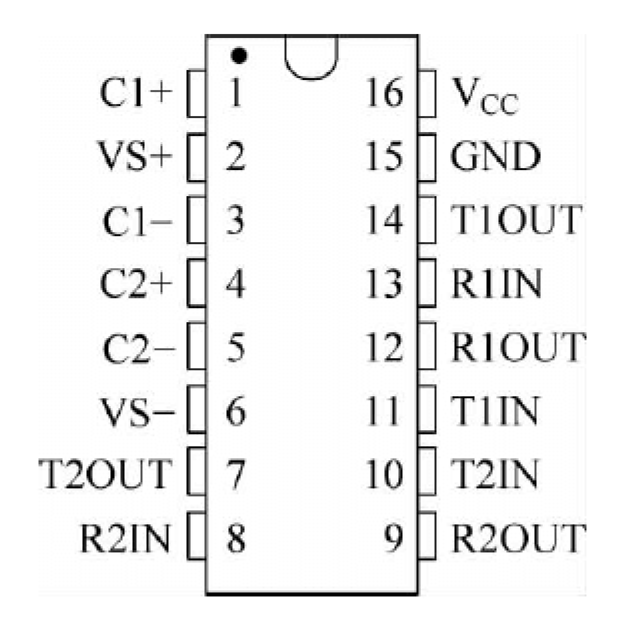


图 6-3　MAX232 引脚

焊接到 PCB 板上的 MAX232 芯片检测方法: 正常情况下,① T1IN=5V,则 T1OUT=−9V;T1IN=0V,则 T1OUT=9V;② 将 R1IN 与 T1OUT 相连,令 T1IN=5V,则 R1OUT=5V;令 T1IN=0V,则 R1OUT=0V。

表 6-2　MAX232 芯片输入输出引脚分类与基本接法

组别	TTL 电平引脚	方向	典 型 接 口	232 电平引脚	方向	典 型 接 口
1	11(T1IN)	输入	接 MCU 的 TxD	13(R1IN)	输入	接到 9 芯接口的 3 脚 RxD
	12(R1OUT)	输出	接 MCU 的 RxD	14(T1OUT)	输出	接到 9 芯接口的 2 脚 TxD
2	10(T2IN)	输入	接 MCU 的 TxD	8(R2IN)	输入	接到 9 芯接口的 3 脚 RxD
	9(R2OUT)	输出	接 MCU 的 RxD	7(T2OUT)	输出	接到 9 芯接口的 2 脚 TxD

具有串行通信接口的 MCU，一般具有发送引脚(TxD)与接收引脚(RxD)，不同公司或不同系列的 MCU，使用的引脚缩写名可能不一致，但含义相同。串行通信接口的外围硬件电路，主要目的是将 MCU 的发送引脚 TxD 与接收引脚 RxD 的 TTL 电平，通过 RS232 电平转换芯片转换为 RS232 电平。图 6-4 给出了基本串行通信接口的电平转换电路。进行 MCU 的串行通信接口编程时，只针对 MCU 的发送与接收引脚，与 MAX232 无关，MAX232 只是起到电平转换作用。MAX232 芯片进行电平转换的基本原理如下。

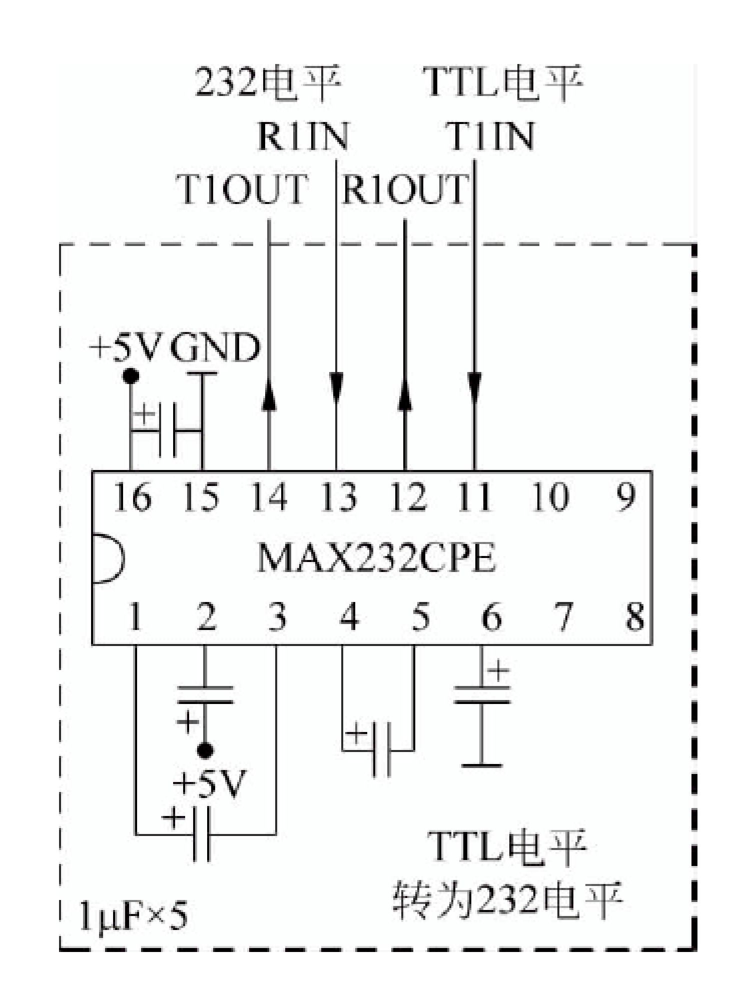


图 6-4　串行通信接口电平转换电路

发送过程： MCU 的 TxD(TTL 电平)经过 MAX232 的 11 脚(T1IN)送到 MAX232 内部，在内部 TTL 电平被“提升”为 232 电平，通过 14 脚(T1OUT)发送出去。

接收过程： 外部 232 电平经过 MAX232 的 13 脚(R1IN)进入到 MAX232 的内部，在内部 232 电平被“降低”为 TTL 电平，经过 12 脚(R1OUT)送到 MCU 的 RxD，进入 MCU 内部。

随着 USB 接口的普及，9 芯串口正在逐渐从 PC，特别是从便携式电脑上消失。于是出现 232-USB 转换线、TTL-USB 转换线，在 PC 上安装相应的驱动软件，就可在 PC 上使用一般的串行通信编程方式，通过 USB 接口实现与 MCU 的串行通信。

6.1.4　串行通信编程模型

从基本原理角度看，串行通信接口 UART 的主要功能是：接收时，把外部的单线输入的数据变成一个字节的并行数据送入 MCU 内部；发送时，把需要发送的一个字节的并行数据转换为单线输出。图 6-5 给出了一般 MCU 的 UART 模块的功能描述。为了设置波特率 UART 应具有波特率寄存器。为了能够设置通信格式、是否校验、是否允许中断等，UART 应具有控制寄存器。而要知道串口是否有数据可收、数据是否发送出去等，需要有 UART 状态寄存器。当然，若一个寄存器不够用，控制与状态寄存器可能有多个。而 UART 数据寄存器存放要发送的数据，也存放接收的数据，这并不冲突，因为发送与接收的实际工作是通过“发送移位寄存器”和“接收移位寄存器”完成的。编程时，程序员并不直接与“发送移位寄存器”和“接收移位寄存器”打交道，只与数据寄存器打交道，所以 MCU 中并没有设置“发送移位寄存器”和“接收移位寄存器”的映像地址。发送时，程序员通过判定状态寄存器的相应位，了解是否可以发送一个新的数据。若可以发送，则将待发送的数据放入“UART 数据寄存器”中就可以了，剩下的工作由 MCU 自动完成：将数据从“UART 数据寄存器”送到“发送移位寄存器”，硬件驱动将“发送移位寄存器”的数据一位一位地按照规定的波特率移到发送引脚 TxD，供对方接收。接收时，数据一位一位地从接收引脚 RxD 进入“接收移位寄存器”，当收到一个完整字节时，MCU 会自动将数据送入“UART 数据寄存器”，并将状态寄存器的相应位改变，供程序员判定并取出数据。

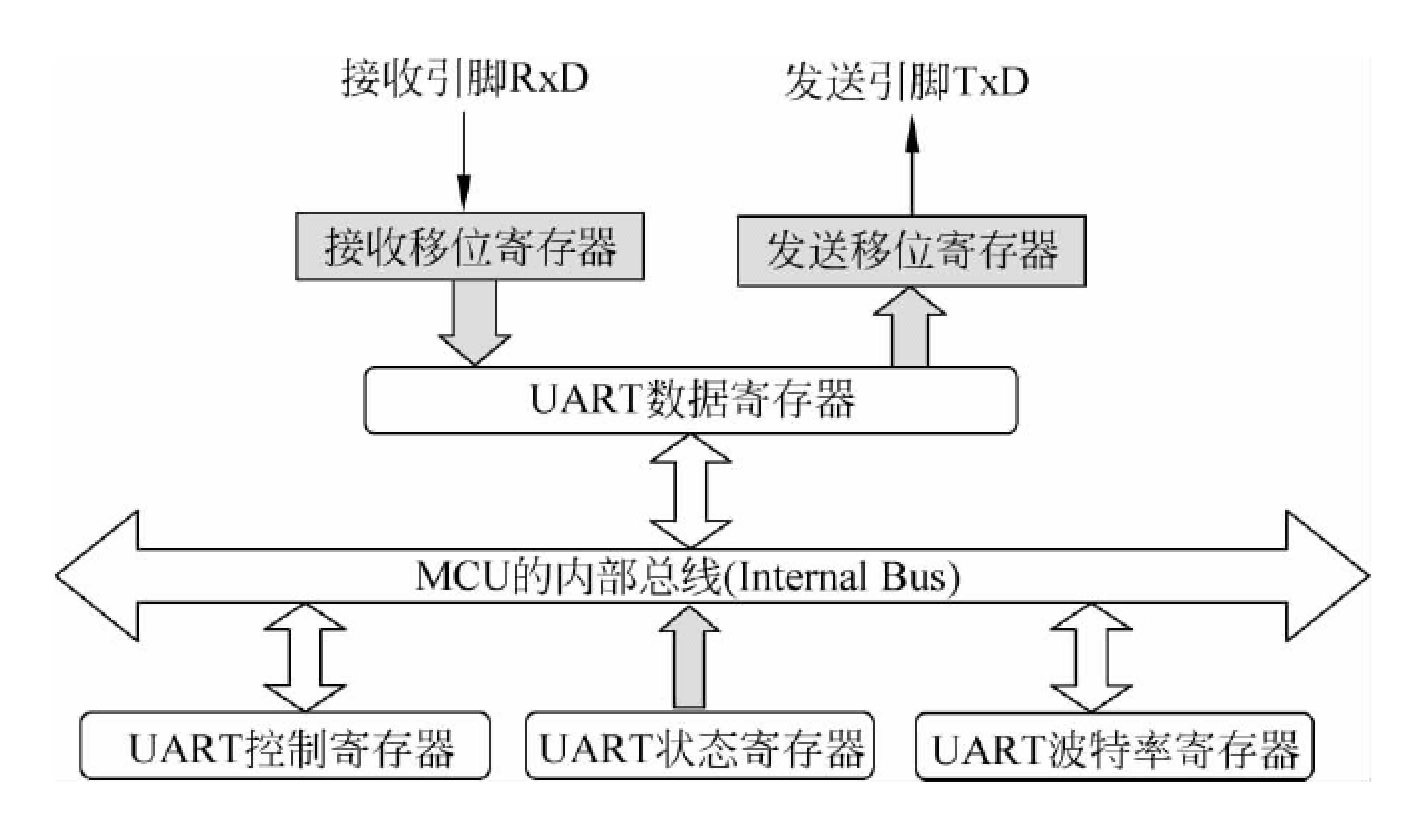


图 6-5 UART 编程模型

6.2 KL25/26 芯片 UART 驱动构件及使用方法

6.2.1 KL25/26 芯片 UART 引脚

KL25/26 中共有三个 UART 模块，分别为 UART0、UART1 和 UART2。每个 UART 的发送数据引脚为 UARTx_TX，接收数据引脚为 UARTx_RX。“x”表示串口模块编号，取值为 0～2。它们并不是固定在哪几个引脚上，而是通过系统集成模块 SIM 提供的引脚选择寄存器编程来设定。根据附录 A(KL25/26 的引脚功能分配)，KL25 与 KL26 中可以配置为串口的引脚见表 6-3。

表 6-3 KL25/KL26 的串口引脚及默认使用的引脚

引脚号		引脚名	ALT2	ALT3	ALT4	备注
KL25	KL26					
1	1	PTE0		UART1_TX		KL25 板用
2	2	PTE1		UART1_RX		
13	9	PTE20			UART0_TX	
14	10	PTE21			UART0_RX	
15	11	PTE22			UART2_TX	KL25 板用
16	12	PTE23			UART2_RX	
27	23	PTA1	UART0_RX			
28	24	PTA2	UART0_TX			
34	(无)	PTA14		UART0_RX		KL25 板用
35	(无)	PTA15		UART0_TX		KL26 板无
40	32	PTA18		UART1_RX		
41	33	PTA19		UART1_TX		
51	39	PTB16		UART0_RX		
52	40	PTB17		UART0_TX		
56	46	PTC3		UART1_RX		

续表

引　脚　号		引脚名	ALT2	ALT3	ALT4	备　注
KL25	KL26					
58	49	PTC4		UART1_TX		
75	59	PTD2		UART2_RX		
76	60	PTD3		UART2_TX		
77	61	PTD4		UART2_RX		
78	62	PTD5		UART2_TX		
79	63	PTD6		UART0_RX		
80	64	PTD7		UART0_TX		

6.2.2　UART 驱动构件基本要素分析与头文件

UART 驱动构件由头文件 uart.h 及源代码文件 uart.c 组成，放入 uart 文件夹中，供应用程序开发调用。

UART 具有初始化、发送和接收三种基本操作。下面分析串口初始化函数的参数应该有哪些。首先应该有串口号，因为一个 MCU 有若干串口，必须确定使用哪个串口；其次是波特率，因为必须确定使用什么速度收发。至于波特率使用哪个时钟来产生，这并不重要，这里确定使用系统总线时钟，就不需要传入这个参数了。关于奇偶校验，由于实际使用主要是多字节组成的一个帧，自行定义通信协议，单字节校验意义不大；此外，串口在嵌入式系统中的重要作用是实现类似 C 语言中的 printf 函数功能，也不宜使用单字节校验，因此就不校验。这样，串口初始化函数就有两个参数：串口与波特率。但是，KL 系列 MCU 的一个串口，可以在不同引脚组上，实际应用中使用哪个引脚组，应该是在应用开发板硬件设计阶段就确定的，为了使驱动构件适应这个场景，可在头文件中使用"宏"进行定义，确定使用的引脚组。这个方法也有缺点，就是若把源代码文件编译成库，再修改宏定义就不起作用了，必须重新使用源程序进行编译，这是所有宏定义的共性。

从知识要素角度，进一步分析 UART 驱动构件的基本函数，与寄存器直接打交道的有：初始化、发送单个字节与接收单个字节的函数，以及使能及禁止接收中断、获取接收中断状态的函数。发送中断不具有实际应用价值，可以忽略。

通过以上简明分析，串口驱动构件可封装下列 9 个基本功能函数。

(1) 串口初始化：void uart_init(uint_8 uartNo, uint_32 baud_rate)；

(2) 发送单个字节：uint_8 uart_send1(uint_8 uartNo, uint_8 ch)；

(3) 发送 N 个字节：uint_8 uart_sendN(uint_8 uartNo ,uint_16 len,uint_8 * buff)；

(4) 发送字符串：uint_8 uart_send_string(uint_8 uartNo, void * buff)；

(5) 接收单个字节：uint_8 uart_re1(uint_8 uartNo,uint_8 * fp)；

(6) 接收 N 个字节：uint_8 uart_reN(uint_8 uartNo,uint_16 len,uint_8 * buff)；

(7) 使能串口接收中断：uart_enable_re_int(uint_8 uartNo)；

(8) 禁止串口接收中断：uart_disable_re_int(uint_8 uartNo)；

(9) 获取接收中断状态：uart_get_re_int(uint_8 uartNo)。

下面给出 UART 驱动构件的头文件(uart.h)。在头文件中用宏定义方式统一了使用

的串口号(UART_0、UART_1、UART_2),以及实际使用时它们所在的引脚组。

```
//==========================================================================
//文件名称: uart.h
//功能概要: UART 底层驱动构件头文件
//版权所有: 苏州大学嵌入式中心(sumcu.suda.edu.cn)
//更新记录: 2015-05-11 V1.0, 2016-05-01 V6.0(WYH)
//适用芯片: KL25、KL26(使用时,注意是否存在实际引脚)
//==========================================================================

#ifndef _UART_H                          //防止重复定义(开头)
#define _UART_H

#include "common.h"                      //包含公共要素头文件

//宏定义串口号
#define UART_0 0
#define UART_1 1
#define UART_2 2
//配置 UARTx 使用的引脚组
//UART_0 的引脚组配置: 0:PTE20~21, 1:PTA1~2, 2:PTA14~15, 3:PTB16~17, 4:PTD6~7
#define UART_0_GROUP 2
//UART_1 的引脚组配置: 0:PTE0~1, 1:PTA18~19, 2:PTC3~4
#define UART_1_GROUP 0
//UART_2 的引脚组配置: 0:PTE22~23, 1:PTD2~3, 2:PTD4~5
#define UART_2_GROUP 0
//==========================================================================
//函数名称: uart_init
//功能概要: 初始化 uart 模块
//参数说明: uartNo:串口号: UART_0、UART_1、UART_2
//          baud:波特率: 300、600、1200、2400、4800、9600、19200、115200...
//函数返回: 无
//==========================================================================
void uart_init(uint_8 uartNo, uint_32 baud_rate);

//==========================================================================
//函数名称: uart_send1
//参数说明: uartNo: 串口号:UART_0、UART_1、UART_2
//          ch:要发送的字节
//函数返回: 函数运行状态: 1=发送成功; 0=发送失败
//功能概要: 串行发送 1 个字节
//==========================================================================
uint_8 uart_send1(uint_8 uartNo, uint_8 ch);

//==========================================================================
//函数名称: uart_sendN
//参数说明: uartNo: 串口号:UART_0、UART_1、UART_2
//          buff: 发送缓冲区
//          len:发送长度
//函数返回: 函数运行状态: 1=发送成功; 0=发送失败
//功能概要: 串行发送 n 个字节
//==========================================================================
```

```
uint_8 uart_sendN(uint_8 uartNo,uint_16 len,uint_8 * buff);

//==========================================================
//函数名称: uart_send_string
//参数说明: uartNo:UART 模块号:UART_0、UART_1、UART_2
//          buff:要发送的字符串的首地址
//函数返回: 函数运行状态: 1=发送成功; 0=发送失败
//功能概要: 从指定 UART 端口发送一个以'\0'结束的字符串
//==========================================================
uint_8 uart_send_string(uint_8 uartNo, void * buff);

//==========================================================
//函数名称: uart_re1
//参数说明: uartNo: 串口号:UART_0、UART_1、UART_2
//          * fp:接收成功标志的指针: * fp=1:接收成功; * fp=0:接收失败
//函数返回: 接收返回字节
//功能概要: 串行接收 1 个字节
//==========================================================
uint_8 uart_re1(uint_8 uartNo,uint_8 * fp);

//==========================================================
//函数名称: uart_reN
//参数说明: uartNo: 串口号:UART_0、UART_1、UART_2
//          buff: 接收缓冲区
//          len: 接收长度
//函数返回: 函数运行状态 1=接收成功;0=接收失败
//功能概要: 串行接收 n 个字节,放入 buff 中
//==========================================================
uint_8 uart_reN(uint_8 uartNo,uint_16 len,uint_8 * buff);

//==========================================================
//函数名称: uart_enable_re_int
//参数说明: uartNo: 串口号:UART_0、UART_1、UART_2
//函数返回: 无
//功能概要: 开串口接收中断
//==========================================================
void uart_enable_re_int(uint_8 uartNo);

//==========================================================
//函数名称: uart_disable_re_int
//参数说明: uartNo: 串口号:UART_0、UART_1、UART_2
//函数返回: 无
//功能概要: 关串口接收中断
//==========================================================
void uart_disable_re_int(uint_8 uartNo);

//==========================================================
//函数名称: uart_get_re_int
//参数说明: uartNo: 串口号:UART_0、UART_1、UART_2
//函数返回: 接收中断标志 1=有接收中断;0=无接收中断
```

```
//功能概要：获取串口接收中断标志，同时禁用发送中断
//===========================================================
uint_8 uart_get_re_int(uint_8 uartNo);

#endif                    //防止重复定义(结尾)
```

由于涉及中断方式编程，将在随后介绍“printf 的设置方法与使用”之后，给出 ARM Cortex-M0+非内核模块中断编程结构，然后给出串口接收中断编程举例(6.3.3节)。

6.2.3　printf 的设置方法与使用

除了使用 UART 驱动构件中封装的 API 函数之外，还可以使用格式化输出函数 printf 来灵活地从串口输出调试信息，配合 PC 或笔记本上的串口调试工具，可方便地进行嵌入式程序的调试。printf 函数的实现在工程目录..\07_App_Component\printf\printf.c 文件中，同文件夹下的 printf.h 头文件则包含 printf 函数的声明，若要使用 printf 函数，可在工程的总头文件..\08_Source\includes.h 中将 printf.h 包含进来，以便其他文件使用。

在使用 printf 函数之前，需要先进行相应的设置来将其与希望使用的串口模块关联起来。设置步骤如下。

(1) 在 printf 头文件..\07_App_Component\printf\printf.h 中宏定义需要与 printf 相关联的调试串口号，例如：

```
#define UART_Debug UART_2          //printf 函数使用的串口号
```

(2) 在使用 printf 前，调用 UART 驱动构件中的初始化函数对使用的调试串口进行初始化，配置其波特率。例如：

```
uart_init(UART_Debug,9600);          //初始化"调试串口"
```

这样就将相应的串口模块与 printf 函数关联起来了。

关于 printf 函数的使用方法，请参见附录 C 的介绍。

6.3　ARM Cortex-M0+中断机制及 KL25/26 中断编程步骤

6.3.1　关于中断的通用基础知识

1. 中断的基本概念

1) 中断与异常的基本含义

异常(Exception)是 CPU 强行从正常的程序运行切换到由某些内部或外部条件所要求的处理任务上去，这些任务的紧急程度优先于 CPU 正在运行的任务。引起异常的外部条件通常来自外围设备、硬件断点请求、访问错误和复位等；引起异常的内部条件通常为指令、不对界错误、违反特权级和跟踪等。一些文献把硬件复位和硬件中断都归类为**异常**，把

硬件复位看作是一种具有**最高优先级的异常**，而把来自 CPU 外围设备的强行任务切换请求称为**中断**(Interrupt)，软件上表现为将程序计数器(PC)指针强制转到中断服务程序入口地址运行。

CPU 在指令流水线的译码或者运行阶段识别异常。若检测到一个异常，则强行中止后面尚未达到该阶段的指令。对于在指令译码阶段检测到的异常，以及对于与运行阶段有关的指令异常来说，由于引起的异常与该指令本身无关，指令并没有得到正确运行，所以为该类异常保存的程序计数器 PC 的值指向引起该异常的指令，以便异常返回后重新运行。对于中断和跟踪异常(异常与指令本身有关)，CPU 在运行完当前指令后才识别和检测这类异常，故为该类异常保存的 PC 值是指向要运行的下一条指令。

CPU 对复位、中断、异常具有同样的处理过程，本书随后在谈及这个处理过程时**统称为中断**。

2）中断源、中断向量表与中断向量号

可以引起 CPU 产生中断的外部器件被称为**中断源**。一个 CPU 通常可以识别多个中断源，每个中断源产生中断后，分别要运行相应的中断服务例程(Interrupt Service Routine，ISR)，这些中断服务例程 ISR 的起始地址(叫作**中断向量地址**)放在一段连续的存储区域内，这个存储区被称为**中断向量表**。实际上，中断向量表是一个指针数组，内容是中断服务例程 ISR 的首地址。

给 CPU 能够识别的每个中断源编个号，就叫**中断向量号**。通常情况下，在程序书写时，中断向量表按中断向量号从小到大的顺序填写中断服务例程 ISR 的首地址，不能遗漏。即使某个中断不需要使用，也要在中断向量表对应的项中填入默认中断服务例程 ISR 的首地址，因为中断向量表是连续存储区，与连续的中断向量号相对应。默认中断服务例程 ISR 的内容，一般为直接返回语句，即没有任何功能。默认中断服务例程 ISR 的存在，不仅是给未用中断的中断向量表项“补白”使用，也可以防止未用中断误发生后有个去处，就直接返回原处。

3）中断服务例程 ISR

中断提供了一种机制，来打断当前正在运行的程序，并且保存当前 CPU 状态(CPU 内部寄存器)，转而去运行一个中断处理程序，然后恢复 CPU 状态，以便恢复 CPU 到运行中断之前的状态，同时使得中断前的程序得以继续运行。中断时打断当前正在运行的程序，而转去运行的一个中断处理程序，通常被称为**中断服务例程**(Interrupt Service Routine，ISR)，也被称为**中断处理函数**。

4）中断优先级、可屏蔽中断和不可屏蔽中断

在进行 CPU 设计时，一般定义了中断源的优先级。若 CPU 在程序运行过程中，有两个以上中断同时发生，则优先级最高的中断得到最先响应。

根据中断是否可以通过程序设置的方式被屏蔽，可将中断划分为可屏蔽中断和不可屏蔽中断两种。**可屏蔽中断**是指可通过程序设置的方式决定不响应该中断，即该中断被屏蔽了。**不可屏蔽中断**是指不能通过程序方式关闭的中断。

2. 中断处理的基本过程

中断处理的基本过程分为中断请求、中断检测、中断响应与中断处理等过程。

1）中断请求

当某一中断源需要 CPU 为其服务时，它将会向 CPU 发出中断请求信号(一种电信

号)。中断控制器获取中断源硬件设备的中断向量号[①],并通过识别的中断向量号将对应硬件中断源模块的中断状态寄存器中的"中断请求位"置位,以便CPU知道何种中断请求来了。

2) 中断采样(检测)

CPU在每条指令结束的时候将会检查中断请求或者系统是否满足异常条件,为此,多数CPU专门在指令周期中使用了中断周期。在中断周期中,CPU将会检测系统中是否有中断请求信号,若此时有中断请求信号,则CPU将会暂停当前运行的任务,转而去对中断事件进行响应,若系统中没有中断请求信号则继续运行当前任务。

3) 中断响应与中断处理

中断响应的过程是由系统自动完成的,对于用户来说是透明的操作。在中断的响应过程中,首先CPU会查找中断源所对应的模块中断是否被允许,若被允许,则响应该中断请求。中断响应的过程要求CPU保存当前环境的"上下文(Context)"于堆栈中。通过中断向量号找到中断向量表中对应的中断服务例程ISR,转而去运行中断处理服务ISR。中断处理术语中,简单地理解"上下文"即指CPU内部寄存器,其含义是在中断发生后,由于CPU在中断服务例程中也会使用CPU内部寄存器,所以需要在调用ISR之前,将CPU内部寄存器保存至指定的RAM地址(栈)中,在中断结束后再将该RAM地址中的数据恢复到CPU内部寄存器中,从而使中断前后程序的"运行现场"没有任何变化。

6.3.2 ARM Cortex-M0+非内核模块中断编程结构

ARM Cortex-M0+把中断分为内核中断与非内核模块中断,第3章中的表3-6给出了KL25/26的中断源,中断向量号内核中断与非内核模块中断统一编号(0~47),非内核中断的中断请求(Interrupt Request)号,简称IRQ中断号,从0至31编号,对应于中断向量号的16~47。

1. M0+中断结构及中断过程

M0+中断结构原理图如图6-6所示。由M0+内核、嵌套中断向量控制器(Nested Vectored Interrupt Controller,NVIC)及模块中断源组成。其中断过程分为两步,第一步,模块中断源向嵌套中断向量控制器NVIC发出中断请求信号;第二步,NVIC对发来的中断信号进行管理,判断该模块中断是否被使能,若使能,通过私有外设总线(Private Peripheral Bus,PPB)发送给M0+内核,由内核进行中断处理。如果同时有多个中断信号到来,NVIC根据设定好的中断优先级进行判断,优先级高的中断首先响应,优先级低的中断挂起,压入堆栈保存;如果优先级完全相同的多个中断源同时请求,则先响应IRQ号较小的,其他的被挂起。例如,当IRQ4[②]的优先级与IRQ5的优先级相等时,IRQ4会比IRQ5先得到响应。

2. M0+嵌套中断向量控制器NVIC内部寄存器简介

M0+嵌套中断向量控制器NVIC内含12个寄存器,如表6-4所示。下面分别进行简介。

① 设备与中断向量号可以不是一一对应的,如果一个设备可以产生多种不同中断,允许有多个中断向量号。

② IRQ中断号为n,简记为IQRn。



图 6-6　M0＋中断结构原理图

表 6-4　嵌套中断向量控制器 NVIC 内各寄存器地址与名称

地　址	名　称	描　述
E000_E100	NVIC_ISER	中断使能寄存器(W/R)
E000_E180	NVIC_ICER	中断禁止寄存器(W/R)
E000_E200	NVIC_ISPR	中断挂起寄存器(W/R)
E000_E280	NVIC_ICPR	清除挂起寄存器(W/R)
E000_E400～E000_E41C	NVIC_IPR0～NVIC_IPR7	优先级寄存器

1）中断使能寄存器(NVIC_ISER)

中断使能寄存器 NVIC_ISER 的 32 位分别对应 32 个外设中断 IRQ 中断号。读取第 n(0～31)位,为 0,表明该中断处于禁止状态；为 1,则处于使能状态。对第 n 位写 1,使能相应 IRQ 号中断,写 0 无效。

2）中断禁止寄存器(NVIC_ICER)

中断禁止寄存器 NVIC_ICER 的 32 位分别对应 32 个外设中断 IRQ 中断号。读取第 n(0～31)位,为 0,表明该中断处于禁止状态；为 1,则处于使能状态。对第 n 位写 1,禁止相应 IRQ 号中断,写 0 无效。

在编写中断程序中,想要使能一个中断,需要将 NVIC_ISER 中的对应位置 1；想要对某个中断相应进行禁止,需要写 1 到 NVIC_ICER 对应的位。

3）挂起/清除挂起寄存器(NVIC_ISPR/NVIC_ICPR)

当中断发生时,正在处理同级或者高优先级异常,或者该中断被屏蔽,则中断不能立即得到响应,此时中断被挂起。中断的挂起状态可以通过中断挂起寄存器(NVIC_ISPR)和清除挂起寄存器(NVIC_ICPR)来读取,还可以通过写这些寄存器进行挂起中断。这里挂起表示排队等待的意思,清除挂起表示取消此次中断请求。

4）优先级寄存器(NVIC_IPR0-NVIC_IPR7)

可以通过优先级寄存器设置非内核中断源的优先级。优先级寄存器(Interrupt Priority Register,IPR)共有 8 个：IPR0～IPR7,每一个优先级寄存器对应 4 个非内核中断源。IPR0 中设置 IRQ0～3 非内核中断优先级,IPR1 中设置 IRQ4～7 非内核中断优先级……IPR7 中设置 IRQ28～31 非内核中断优先级。

由于每一个优先级寄存器 IPR 只对应 4 个非内核中断源,因此每个中断源在 IPR 寄存器中占两位,其他位未用。例如,IPR0 的含义如下。

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	IRQ3		0	0	0	0	0	0	IRQ2		0	0	0	0	0	0	IRQ1		0	0	0	0	0	0	IRQ0		0	0	0	0	0	0
W																																

3. 非内核中断初始化设置步骤

根据本节给出的ARM Cortex-M0+非内核模块中断编程结构，想让一个非内核中断源能够得到内核响应(或禁止)，基本步骤如下。

(1) 设置模块中断使能位使能模块中断，使模块能够发送中断请求信号。例如，在UART中，将控制寄存器C2的RIE位置1。

(2) 查找芯片中断源表(例如表3-6)，找到对应IRQ号，设置嵌套中断向量控制器的中断使能寄存器(NVIC_ISER)，使该中断源对应位置1，允许该中断请求。反之，若要禁止该中断，则设置嵌套中断向量控制器的中断禁止寄存器(NVIC_ICER)，使该中断源对应位置1即可。

(3) 若要设置其优先级，可对优先级寄存器编程。

本书网上教学资源，已经在各外设模块底层驱动构件中封装了模块中断使能与禁止的函数，可直接使用。这里阐述的目的是为了使读者理解其中的编程原理。读者只要选择一个含有中断的构件，理解其使能中断与禁止中断函数即可。

6.3.3 KL25/26中断编程步骤——以串口接收中断为例

3.4节给出了KL25/26的中断源(表3-6)及中断向量表。现在来看看如何进行一个中断的编程。下面以串口2接收中断为例，阐述KL25/26中断编程步骤。

1. 先导工作——在UART驱动构件头文件(uart.h)中设定串口2使用哪组引脚

在头文件uart.h中，设定UART_2实际使用的引脚。宏定义UART_x_GROUP(x=0～2)确定UARTx实际使用的引脚，例如，若UART2实际使用的引脚分别是PTE22和PTE23，则设置：

```
//UART_2的引脚组配置：0:PTE22～23，1:PTD2～3，2:PTD4～5
#define UART_2_GROUP    0
```

2. main.c文件中的工作——串口初始化、使能模块中断、开总中断

首先查看uart构件的头文件uart.h，看看串口2的符号表达。经过查看宏定义，知道串口2的符号表达为"UART_2"，可以作为调用uart构件的实参使用。随后在main.c文件开始处进行以下编程。

(1) 在"初始化外设模块"位置调用uart构件中的初始化函数。

```
uart_init(UART_2, 9600);        //波特率使用9600
```

(2) 在"初始化外设模块"位置调用uart构件中的使能模块中断函数。

```
uart_enable_re_int(UART_2);          //使能串口2接收中断
```

(3) 在"开总中断"位置调用common.h文件中的开总中断宏函数。

```
ENABLE_INTERRUPTS;           //开总中断
```

这样，串口2接收中断初始化完成。

3. 在 startup_MKL25Z4.S 文件的中断向量表中找到相应中断服务例程的函数名

在中断向量表中找到串口 2 接收中断服务例程的函数名是 UART2_IRQHandler。

4. 在"..\08_Source\isr.c"进行中断功能的编程

紧接着，可以在"..\08_Source\isr.c"文件中添加函数：

```
void UART2_IRQHandler(void)
{
}
```

就可在此处进行串口 2 接收中断功能的编程了。这里的函数会取代原来的默认函数。这样就避免了用户直接对中断向量表进行修改，而 startup_MKL25Z4.S 文件中采用"弱定义"的方式为用户提供编程接口，既方便用户使用，同时也提高了系统编程的安全性。

5. 中断服务程序设计例

中断服务程序的设计与普通构件函数设计是一样的，只是这些程序只有在中断产生时才被运行。为了规范编程，统一将各个中断服务程序，放在工程框架中的"..\08_Source\isr.c"文件中。如编写一个串口 2 接收中断服务程序，当串口 2 有一个字节的数据到来时产生接收中断，将会执行 UART2_IRQHandler 函数。这个程序首先进入临界区[①]关总中断，接收一个到来的字符。若接收成功，则把这个字符发送回去，退出临界区。

```
//================中断函数服务例程==================
//串口 2 接收中断服务例程
void UART2_IRQHandler(void)
{
    uint_8 ch, flag;
    flag = 1
    DISABLE_INTERRUPTS;                    //关总中断
    ch = uart_re1(UART_2, &flag);          //调用接收一个字节的函数
    if (0 == flag)                         //若收到一个字节
    {
        uart_send1(UART_2, ch);            //向原串口发回一个字节
    }
    ENABLE_INTERRUPTS;                     //开总中断
}
```

测试程序在网上教学资源的"..\02-Software\program\ch06-UART"文件夹中。由于通信涉及两方，为了更好地掌握串行通信编程，**该文件夹还给出了 PC 方 C# 串口测试源程序。掌握一门可以与 MCU 通信的 PC 编程语言，并合理地加以应用，对嵌入式系统的学习将有很大帮助。**

① 有些情况下，一些程序段是需要连续执行而不能被打断的，此时，程序对 CPU 资源的使用是独占的，此时称为"临界状态"，不能被打断的过程称为对"临界区"的访问。为防止在执行关键操作时被外部事件打断，一般通过关中断的方式使程序访问临界区，屏蔽外部事件的影响。执行完关键操作后退出临界区，打开中断，恢复对中断的响应能力。

6.4　UART驱动构件的设计方法

设计UART驱动构件需要深入理解UART模块编程结构(即UART模块的映像寄存器),还要掌握基本编程过程与调试方法,这是一项细致且有一定难度的工作。这节内容可由教师根据教学基本要求进行取舍。

6.4.1　UART模块编程结构

以下寄存器的用法在KL25/26的芯片手册上有详细的说明,下面按初始化顺序阐述基本编程需要使用的寄存器。注意,KL25与KL26的UART模块在使用上没有区别,下面所列寄存器名中的"x"表示UART模块编号,取0～2。

1. 寄存器地址分析

KL25/26芯片有三个UART模块。每个模块有其对应的寄存器。以下地址分析均为十六进制,为书写简化起见,在不致引起歧义的情况下,略去十六进制后缀"0x"不写。

UART模块x的寄存器的地址＝4006_A000＋x×1000＋n×1(x＝0～2;模块0中n＝0～B,模块1、2中n＝0～8,n代表寄存器号)。

2. 控制寄存器

1) UARTx控制寄存器2(UARTx_C2)

UARTx控制寄存器2(UARTx_C2)主要用于收/发及相关中断控制设置,如表6-5所示。

表6-5　UARTx_C2结构

数据位	D7	D6	D5	D4	D3	D2	D1	D0
读/写	TIE	TCIE	RIE	ILIE	TE	RE	RWU	SBK
复位	0							

D7(TIE)——发送中断使能位。与状态寄存器中的TDRE配合使用。TIE＝0,发送中断禁用(使用轮询);TIE＝1,当TDRE＝1时,发生中断请求。在实际的工程项目中发送中断一般不使用,因此本构件在初始化函数中会将该位清0禁用发送中断。

D6(TCIE)——发送完成中断使能位,与状态寄存器中的TC位配合使用。TCIE＝0,TC对应的中断禁用(使用轮询);TCIE＝1,当TC＝1时,发生中断请求。

D5(RIE)——接收中断使能位。与状态寄存器中的RDRF配合使用。RIE＝0,RDRF中断禁止(使用轮询);RIE＝1,当RDRF＝1时,发生中断请求。

D4(ILIE)——空闲线中断使能,与状态寄存器中的IDLE配合使用。ILIE＝0,IDLE中断禁止(使用轮询);ILIE＝1,当IDLE＝1时,发生中断请求。

D3(TE)——发射器使能位。TE必须是1来使用UART发送器。通常在TE＝1时,UART_TX引脚作为UART系统的输出。当UART配置为单线模式(LOOPS＝1且RSRC＝1)时,UART_C3中的TXDIR位将控制单线模式下UART_TX引脚的通信方向。通过对TE位先写TE＝0然后写TE＝1,TE位也可以对空闲字符排队。当TE为0时,发

送器一直控制端口 UART_TX 引脚，直到完成任何数据、等待空闲或等待中止符的传输，才允许引脚为三态。TE=0，发送器禁止；TE=1，发送器使能。

D2(RE)——接收器使能。当 UART 接收器关闭或 LOOPS 被置位，UART 不使用 UART_RX 脚。当 RE 被写 0，接收机完成接收的当前字符(如有的话)。RE=0，接收器禁止；RE=1，接收器使能。

D1(RWU)——接收器唤醒控制位。向该位写 1 可将接收器设置为待机状态，等待对选中的条件进行自动检测。唤醒方式有空闲线唤醒(WAKE=0)和地址位唤醒(WAKE=1)两种。RWU=0，正常 UART 接收器操作；RWU=1，接收器等待唤醒条件。

D0(SBK)——发送中止使能位。在发送数据流中，写一个 1 然后一个 0 到 SBK 队列的中止字符。如果 BRK13=1 时，只要 SBK 被设定，其他中止字符中的 10～13，或 13～16，逻辑 0 的位时间排队。根据 SBK 的设置，清除当前正在发送的信息，在软件清除 SBK 前，第二个中止字符可能会排队。SBK=0，正常发送操作；SBK=1，队列中止字符被发送。

2) UARTx 控制寄存器 1(UARTx_C1)

UARTx 控制寄存器 1(UARTx_C1)主要用于设置 SCI 的工作方式，可选择运行模式、唤醒模式、空闲类型检测以及奇偶校验等，如表 6-6 所示。

表 6-6　UARTx_C1 结构

数据位	D7	D6	D5	D4	D3	D2	D1	D0
读/写	LOOPS	DOZEEN/ UARTSWAI	RSRC	M	WAKE	ILT	PE	PT
复位	0							

D7(LOOPS)——循环模式选择。在循环模式和正常的 2 针全双工模式之间选择。当 LOOPS=1，则发送器的输出连接到接收器的输入，可用于循环模式或单线模式，单线模式下，UART 不使用 UART_RX 引脚(见 RSRC)。LOOPS=0，采用不同的引脚正常操作 UART_RX 和 UART_TX(全双工)。

D6(DOZEEN/UARTSWAI)——休眠使能/UART 等待模式停止位。在 UART0 模块中该位为 DOZEEN，DOZEEN=0，UART 在等待模式下继续运行；DOZEEN=1，UART 在等待模式下被禁用。在 UART1 或 UART2 模块中，该位为 UARTSWAI，UARTSWAI=0，UART 时钟在等待模式下仍然运行，这样 UART 就可以作为唤醒 CPU 的中断源。UARTSWAI=1，当 CPU 处于等待模式时，UART 时钟停止。

D5(RSRC)——接收器信号源位。在 LOOPS 置为 1 时，该位有效。当 LOOPS 被置位，接收器的输入在内部连接 UART_TX 引脚，RSRC 决定这个连接是否也被连接到发送器的输出。RSRC=0，选择内部循环模式，UART 不使用 UART_RX 的引脚。RSRC=1，UART 采用单线模式，UART_TX 引脚被连接到发送器的输出和接收器的输入。

D4(M)——数据帧格式选择位(9 位或 8 位模式选择)。M=0，接收和发送使用 8 位数据字符；M=1 接收和发送使用 9 位数据字符。

D3(WAKE)——接收器唤醒模式选择位。WAKE=0 空闲线唤醒；WAKE=1 地址标志唤醒。

D2(ILT)——空闲线类型选择位。ILT=1，在停止位后开始对空闲特征位计数。ILT=

0,在开始位后立即对空闲特征位开始计数。

D1(PE)——奇偶校验使能位。PE=0,奇偶校验禁止。PE=1,奇偶校验使能。当奇偶校验使能,停止位的前一位被视为奇偶校验位。

D0(PT)——奇偶校验类型位。当PE使能(PE=1)时,PT=0偶校验;PT=1奇校验。

3) UART0控制寄存器4(UART0_C4)

UART0_C4结构如表6-7所示。

表6-7　UART0_C4结构

数据位	D7	D6	D5	D4	D3	D2	D1	D0
读/写	MAEN1	MAEN2	M10	OSR				
复位	0				1			

注:UART0_C4,仅仅在模块0中是如此定义,UART模块1、2中与UART0_C4的定义不相同。

D7(MAEN1)——匹配地址模式使能1。MAEN1=0,如果MAEN2被清零,接收的所有数据传送到数据缓冲区;MAEN1=1,接收的所有的最高有效位(MSB)被清零的数据将被丢弃。最高有效位被设置,所有接收到的数据与MA1寄存器内容进行比较。如果匹配失败,数据被丢弃。如果匹配成功,数据被传到数据寄存器中。

D6(MAEN2)——匹配地址模式使能2。MAEN2=0,如果MAEN1被清零,接收的所有数据传送到数据缓冲区;MAEN2=1,最高有效位被清零,接收的所有数据将被丢弃。最高有效位被设置,所有接收到的数据与MA2寄存器内容进行比较。如果匹配失败,数据被丢弃。如果匹配成功,数据被传到数据寄存器中。

D5(M10)——10位模式选择。M10位将导致第10位成为串行传输的一部分。当发送器和接收器都被禁用时,该位可被更改。M10=0,接收器和发送器使用8位或9位的数据字符;M10=1,接收器和发送器使用10位的数据字符。

D4~D0(OSR)——过采样率(过采样就是多次采样对结果求均值,以提高精度)。此字段为接收器配置了4倍(00011)到32倍(11111)之间的过采样率。如果配置的数字不在此范围,将默认为过采样率为16倍(01111)。只有当收发器都禁用时,该字段才可被修改。

4) UART0控制寄存器5(UART0_C5)

UART0_C5结构如表6-8所示。

表6-8　UART0_C5结构

数据位	D7	D6	D5	D4	D3	D2	D1	D0
读/写	TDMAE	0	RDMAE	0			BOTHEDGE	RESYNCDIS
复位	0							

注:UART0_C5,与UART模块1、2中与UART0_C4的定义类似,但也有不同之处。另:UART模块1、2中无控制寄存器5。

D7——TDMAE,发送器DMA使能。TDMAE配置发送数据寄存器空标志,S1[TDRE],以产生DMA请求。TDMAE=0,DMA请求禁止;TDMAE=1,DMA请求允许。

D6——保留位,只读为0。

D5——RDMAE，接收器满 DMA 使能。RDMAE 配置接收器的数据寄存器满标志，S1[RDRF]，以产生 DMA 请求。RDMAE＝0，DMA 请求禁止；RDMAE＝1，DMA 请求允许。

D4～D2——保留位，只读为 0。

D1——BOTHEDGE，双边沿采样。允许在波特率时钟的两个边缘上对接收到的数据采样，有效地加倍接收器对于一个给定的过采样率的输入数据采样的次数。对于 x4 和 x7 的过采样比率，该位必须被设置，对于较高的过采样比率是可选的。只有当收发器禁用时，该位可被修改。BOTHEDGE＝0，接收器利用波特率时钟的上升沿对输入数据进行采样。BOTHEDGE＝1，接收器利用波特率时钟的上升沿和下降沿对输入数据进行采样。

注：UART 模块 1、2 中的控制寄存器 4 中该位保留位，只读为 0。

D0——RESYNCDIS，再同步禁止。当设置时，当一个数据 1 跟随数据 0 过渡被检测到时，接收到的数据字的重新同步禁用。RESYNCDIS＝0，在接收到的数据字允许期间重新同步；RESYNCDIS＝1，在接收到的数据字期间禁用重新同步。

注：UART 模块 1、2 中的控制寄存器 4 中该位保留位，只读为 0。

3. 状态寄存器

UARTx_S1 寄存器为 UART 中断或 DMA 请求提供 MCU 的输入，如表 6-9 所示。这个寄存器也可以由 MCU 进行轮询来检测。可以通过读状态寄存器之后读或写(取决于中断标志类型)UART 数据寄存器来清除标志。其他的指令只要不影响 I/O 处理也可以插入到上述两步中运行。

表 6-9　UARTx_S1 结构

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	TDRE	TC	RDRF	IDLE	OR	NF	FE	PF
复位	1	1	0	0	0	0	0	0

注：UART0 中 D4～D0 写 1 清 0。UART1 与 UART2 中该寄存器只读。

D7——TDRE，发送数据寄存器空标志位。该位在复位之后置 1；或者当一个发送数据从缓冲区转移到发送移位器后，该位置位。为清除 TDRE，当 TDRE＝1 时，应先对该位进行读操作，然后写 UART 数据寄存器。TDRE＝0，发送数据寄存器(缓冲器)已满；TDRE＝1，发送数据寄存器(缓冲器)为空。

D6——TC，发送完成标志位。TC＝0，正在发送；TC＝1，发送完成。TC 位在复位后置为 1；或当 TDRE＝1，且无数据、前导符或中止符正在发送，TC 置为 1。TC＝1 时，可通过先读取 UARTx_S1，然后进行以下任意一种操作，TC 位将自动清除：①向 UARTx_D 寄存器写入数据；②通过向 TE 写 0，然后向 TE 写 1，对一个前导字符排队；③通过向控制寄存器 2 的 SBK 位写 1，对一个中止字符排队。

D5——RDRF，接收数据寄存器已满标志位。当接收缓冲区满了以后，该位置位。注意，为清除 RDRF，应先对该位进行读操作，然后读 UART 数据寄存器。RDRF＝0，接收数据寄存器(缓冲器)空；RDRF＝1，接收数据寄存器(缓冲器)已满。

D4——IDLE，空闲线标志。如果 UART 接收线在活动周期之后的空闲持续一个字符时间，则该位置位。当控制寄存器 1 中的 ILT＝0 时，接收器从开始位计时空闲位时间。因

此,如果接收到的字符全为1,那么这些位的时间加上停止位的时间是接收器检测空闲线的时间。当控制寄存器1中的ITL=1时,接收器从停止位开始计时空闲位时间。停止位和刚发送字符中的任意高电平位的时间不能作为接收器检测空闲线的时间。要清除该标志位,可向该位写1。该位清除后,只有在接收到一个新的字符且RDRF=1时,IDLE才能再次置位,即使接收线在额外的周期内保持空闲状态,IDLE也只置位一次。IDLE=0,没有检测到空闲线路;IDLE=1,检测到空闲线路。

D3——OR,接收器溢出标记位。当一个新的字符准备转移到接收数据寄存器(缓冲器),但以前接收到的字符还未从UARTx_D读取时,OR置位。要清除OR,应先读UARTx_S1中的OR,然后读UARTx_D。OR=0,没有溢出;OR=1,接收溢出(新UART数据丢失)。

D2——NF,噪声标志。在UART接收器中采用了高级采样技术,对每一个接收到的位进行三次采样。如果任意一次采样与其他采样不同,将置位NF。NF=0,未检测到噪声;NF=1,在UARTx_D的接收数据中检测到噪声。

D1——FE,帧错误标志。如果在应该出现停止位的时刻检测到0,则该位置位。要清除FE,可先读UARTx_S1,再读UARTx_D。FE=0,未检测到成帧错误,这不能保证成帧正确;FE=1,成帧错误。

D0——PF,奇偶校验错误标志。当奇偶校验使能(PE=1),且接收到数据的奇偶校验位于期望的奇偶校验值不匹配,该位置位。PF=0,没有奇偶效验错误;PF=1,奇偶校验错误。

4. 波特率寄存器:UARTx_BDH、UARTx_BDL

UARTx_BDH寄存器与UARTx_BDL寄存器一同控制着波特率生成器的分频因子,如表6-10所示。只有当收发器都禁用的情况下,13位[SBR12:SBR0]波特率设置位才可被设置。

表6-10　UARTx_BDH与UARTx_BDL结构

数据位	D7	D6	D5	D4	D3	D2	D1	D0
读/写	LBKDIE	RXEDGIE	SBNS	SBR				
复位	0							

D7(LBKDIE)——LIN中止检测中断使能。LBKDIE=0,UART_S2[LBKDIF]会禁止硬件中断(通过轮询机制)。LBKDIE=1,当UART_S2[LBKDIF]标志为1时,有硬件中断请求。

D6(RXEDGIE)——RX输入有效边沿中断允许位。RXEDGIE=0,UART_S2[RXEDGIF]会禁止硬件中断(通过轮询机制)。RXEDGIE=1,当UART_S2[RXEDGIF]标志为1时,有硬件中断请求。

D5(SBNS)——停止位数选择。SBNS决定了停止位的位数。只有当收发器同时处于禁止情况下该位才可以被改变。SBNS=0,一位停止位;SBNS=1,两位停止位。

D4～D0(SBR)——波特率模数因子。它与波特率低字节寄存器(UARTx_BDL)的8位组合成13位[SBR12:SBR0],统称为BR。它们给波特率生成器设置模数因子值。

UARTx_BDL 与 UARTx_BDH 寄存器一起控制着 UART 波特率生成器的预分频因子。只有当收发器都禁用的情况下，13 位[SBR12：SBR0]波特率设置位（波特率模数因子）才可被设置。8 位 UARTx_BDL 寄存器为波特率设置位的低 8 位[SBR7：SBR0]。复位后 UART_BDL 为非零值，复位后波特率生成器被禁止。

SBR[12：0]中的 13 位统称为 BR。它们给波特率生成器设置模数因子值。当 BR 是 1～8191 时，UART0 波特率＝UART0 时钟/((OSR＋1)×BR)。UART1 与 UART2 波特率＝BUSCKL/(16×BR)。

5. 数据寄存器

UARTx_D(x＝0～2)其实是两个单独的 8 位寄存器，读时会返回只读接收数据寄存器中的内容，写时会写到只写发送数据寄存器。

6.4.2　UART 驱动构件源码

UART 驱动构件存放于工程目录“..\02-Software\KL25 共用驱动\KL25 底层驱动构件”文件夹中，供复制使用，各个工程文件夹下的“..\05_Driver\uart”文件夹中 uart 驱动构件与此一致。UART 驱动构件的实现在源程序文件 uart.c 中。下面给出源程序文件 uart.c 中与寄存器相关的主要函数内容。

```
//==========================================================
//文件名称：uart.c
//功能概要：uart 底层驱动构件源文件
//版权所有：苏州大学恩智浦嵌入式中心(sumcu.suda.edu.cn)
//更新记录：2015-7-14 V1.0
//==========================================================
#include "uart.h"

//=====串口 1、2 号地址映射====
static const UART_MemMapPtr UART_ARR[] = {UART1_BASE_PTR, UART2_BASE_PTR};
//====定义串口 IRQ 号对应表====
static IRQn_Type table_irq_uart[3] = {UART0_IRQn, UART1_IRQn, UART2_IRQn};

//内部函数声明
uint_8 uart_is_uartNo(uint_8 uartNo);

//(函数头注释见头文件)
void uart_init(uint_8 uartNo, uint_32 baud_rate)
{
    //局部变量声明
    uint_16 sbr;
    uint_8 temp;
    UART0_MemMapPtr uartch_0=UART0_BASE_PTR;
                                        //uartch_0 为 UART0_MemMapPtr 类型指针
    UART_MemMapPtr uartch_1_2;          //uartch_1_2 为 UART_MemMapPtr 类型指针

    //判断传入串口号参数是否有误，有误直接退出
    if(!uart_is_uartNo(uartNo)) return;
```

```
    //根据带入参数uartNo,给局部变量uartch_0或uartch_1_2赋值
    if(uartNo==0)
    {
        //UART0选择MCGFLLCLK_kHz=48000KHz时钟源
        SIM_SOPT2 |= SIM_SOPT2_UART0SRC(0x1);
        SIM_SOPT2 |= SIM_SOPT2_PLLFLLSEL_MASK;
#ifdef UART_0_GROUP                                  //依据选择配置对应引脚为UART0
        switch(UART_0_GROUP)
        {
        (把使用的引脚配置成UART功能,程序见网上教学资源)
        case 0:
            PORTE_PCR20 |= PORT_PCR_MUX(0x4);        //使能UART0_TXD
            PORTE_PCR21 |= PORT_PCR_MUX(0x4);        //使能UART0_RXD
            break;
         ...
        SIM_SCGC4 |= SIM_SCGC4_UART0_MASK;           //启动串口0时钟
        //暂时关闭串口0发送与接收功能
        UART0_C2_REG(uartch_0) &= ~(UART0_C2_TE_MASK | UART0_C2_RE_MASK);
        //配置串口工作模式:8位无校验模式
        sbr = (uint_16)((SYSTEM_CLK_KHZ * 1000)/(baud_rate * 16));
        temp = UART0_BDH_REG(uartch_0) & ~(UART0_BDH_SBR(0x1F));
        UART0_BDH_REG(uartch_0) = temp | UART0_BDH_SBR(((sbr & 0x1F00) >> 8));
        UART0_BDL_REG(uartch_0) = (uint_8)(sbr & UART0_BDL_SBR_MASK);

        //初始化控制寄存器、清标志位
        UART0_C4_REG(uartch_0) = 0x0F;
        UART0_C1_REG(uartch_0) = 0x00;
        UART0_C3_REG(uartch_0) = 0x00;
        UART0_MA1_REG(uartch_0) = 0x00;
        UART0_MA2_REG(uartch_0) = 0x00;
        UART0_S1_REG(uartch_0) |= 0x1F;
        UART0_S2_REG(uartch_0) |= 0xC0;
        //禁用串口发送中断
        UART0_C2_REG(uartch_0) &= ~UART0_C2_TIE_MASK;
        //启动发送接收
        UART0_C2_REG(uartch_0) |= (UART0_C2_TE_MASK | UART0_C2_RE_MASK);
        ...
    }
}
//(函数头注释见头文件)
uint_8 uart_send1(uint_8 uartNo, uint_8 ch)
{
    uint_32 t;
    UART0_MemMapPtr uartch_0=UART0_BASE_PTR;          //获取UART0基地址
    UART_MemMapPtr uartch_1_2=UART_ARR[uartNo-1];     //获取UART1或者2基地址

    //判断传入串口号参数是否有误,有误直接退出
    if(!uart_is_uartNo(uartNo))
    {
        return 0;
```

```
    }

    for (t = 0; t < 0xFBBB; t++)                          //查询指定次数
    {
        if(0==uartNo)                                      //判断使用的哪个串口
        {
            //发送缓冲区为空则发送数据
            if ( UART0_S1_REG(uartch_0) & UART0_S1_TDRE_MASK)
            {
                UART0_D_REG(uartch_0) = ch;
                break;
            }
        }
        else
        {
            //发送缓冲区为空则发送数据
            if (UART_S1_REG(uartch_1_2) & UART_S1_TDRE_MASK)
            {
                UART_D_REG(uartch_1_2) = ch;
                break;
            }
        }
    }//end for
    if (t >=0xFBBB)
        return 0;                                          //发送超时,发送失败
    else
        return 1;                                          //成功发送
}

(发送 N 字节,发送字符串函数,略)

//(函数头注释见头文件)
uint_8 uart_re1(uint_8 uartNo,uint_8 * fp)
{
    uint_32 t;
    uint_8 dat;
    UART0_MemMapPtr uartch_0=UART0_BASE_PTR;          //获取 UART0 基地址
    UART_MemMapPtr uartch_1_2=UART_ARR[uartNo-1];   //获取 UART1 或者 2 基地址

    //判断传入串口号参数是否有误,有误直接退出
    if(!uart_is_uartNo(uartNo))
    {
        * fp=0;
        return 0;
    }

    for (t = 0; t < 0xFBBB; t++)                          //查询指定次数
    {
        if(0==uartNo)                                      //判断使用的哪个串口
        {
```

```
            //判断接收缓冲区是否满
            if (UART0_S1_REG(uartch_0) & UART0_S1_RDRF_MASK)
                {
                    dat=UART0_D_REG(uartch_0);              //获取数据,清接收中断位
                    * fp = 1;                               //接收成功
                    break;
                }
            }
            else
            {
                //判断接收缓冲区是否满
                if(UART_S1_REG(uartch_1_2) & UART_S1_RDRF_MASK)
                {
                    dat=UART_D_REG(uartch_1_2);             //获取数据,清接收中断位
                    * fp= 1;                                //接收成功
                    break;
                }
            }
        }//end for
        if(t >=0xFBBB)
        {
            dat = 0xFF;
            * fp = 0;                                       //未收到数据
        }
        return dat;                                         //返回接收到的数据
}

(接收 N 字节函数,略)

//(函数头注释见头文件)
void uart_enable_re_int(uint_8 uartNo)
{
    UART0_MemMapPtr uartch_0=UART0_BASE_PTR;           //获取 UART0 基地址
    UART_MemMapPtr uartch_1_2=UART_ARR[uartNo-1];      //获取 UART1 或者 2 基地址

    //判断传入串口号参数是否有误,有误直接退出
    if(!uart_is_uartNo(uartNo))
    {
        return;
    }

    if(0 == uartNo)
        UART0_C2_REG(uartch_0) |= UART0_C2_RIE_MASK;         //开放 UART 接收中断
    else
        UART_C2_REG(uartch_1_2) |= UART_C2_RIE_MASK;         //开放 UART 接收中断
    NVIC_EnableIRQ(table_irq_uart[uartNo]);                //开中断控制器 IRQ 中断
}

//(函数头注释见头文件)
void uart_disable_re_int(uint_8 uartNo)
```

```
{
    UART0_MemMapPtr uartch_0 = UART0_BASE_PTR;       //获取 UART0 基地址
    UART_MemMapPtr uartch_1_2 = UART_ARR[uartNo-1];  //获取 UART1 或者 2 基地址

    //判断传入串口号参数是否有误,有误直接退出
    if(!uart_is_uartNo(uartNo))
    {
        return;
    }

    if(0==uartNo)
        UART0_C2_REG(uartch_0) &= ~UART0_C2_RIE_MASK;    //禁止 UART 接收中断
    else
        UART_C2_REG(uartch_1_2) &= ~UART_C2_RIE_MASK;    //禁止 UART 接收中断
    NVIC_DisableIRQ(table_irq_uart[uartNo]);
}

//(函数头注释见头文件)
uint_8 uart_get_re_int(uint_8 uartNo)
{
    uint_8 flag;
    UART0_MemMapPtr uartch_0=UART0_BASE_PTR;          //获取 UART0 基地址
    UART_MemMapPtr uartch_1_2=UART_ARR[uartNo-1];     //获取 UART1 或者 2 基地址

    //判断传入串口号参数是否有误,有误直接退出
    if(!uart_is_uartNo(uartNo))
    {
        return 0;
    }

    if(0==uartNo)
    {
        //禁用串口发送中断,防止误中断
        UART0_C2_REG(uartch_0) &= (~UART0_C2_TIE_MASK);
        //获取接收中断标志,需同时判断 RDRF 和 RIE
        flag=UART0_S1_REG(uartch_0) & (UART0_S1_RDRF_MASK);
        return(BGET(UART0_S1_RDRF_SHIFT, flag)
                & BGET(UART0_C2_RIE_SHIFT,UART0_C2_REG(uartch_0)));
    }
    else
    {
        //禁用串口发送中断,防止误中断
        UART_C2_REG(uartch_1_2) &= (~UART_C2_TIE_MASK);
        //获取接收中断标志,需同时判断 RDRF 和 RIE
        flag=UART_S1_REG(uartch_1_2) & (UART_S1_RDRF_MASK);
        return(BGET(UART_S1_RDRF_SHIFT, flag)
                & BGET(UART_C2_RIE_SHIFT,UART_C2_REG(uartch_1_2)));
    }
}
```

```
//----------------------------------------以下为内部函数存放处----------------------------
//==========================================================================
//函数名称: uart_is_uartNo
//函数返回: 1:串口号在合理范围内,0: 串口号不合理
//参数说明: 串口号 uartNo :UART_0、UART_1、UART_2
//功能概要: 为程序健壮性而判断 uartNo 是否在串口数字范围内
//==========================================================================
uint_8 uart_is_uartNo(uint_8 uartNo)
{
    if(uartNo < UART_0 || uartNo > UART_2)
        return 0;
    else
        return 1;
}
//---------------------------------------------内部函数结束-------------------------------
```

小　　结

本章为本书的重点之一,串行通信在嵌入式开发中的特殊地位,通过串行通信接口与PC相连,可以借助PC屏幕进行嵌入式开发的调试。本章另一重要内容阐述中断机理、中断编程的基本方法。至此,1～6章已经囊括学习一个新MCU入门环节的完整要素。后续章节将在此规则与框架下学习各知识模块。本章小结如下。

(1) 给出串口通信的通用基础知识。MCU的串口通信模块UART,在硬件上,一般只需要三根线,分别称为发送线(TxD)、接收线(RxD)和地线(GND);通信方式上,属于单字节通信,是嵌入式开发中重要的打桩调试手段。串行通信数据格式可简要表述为:发送器通过发送一个“0”表示一个字节传输的开始,随后一般是一个字节的8位数据。最后,发送器停止位“1”,表示一个字节传送结束。若继续发送下一字节,则重新发送开始位,开始一个新的字节传送。若不发送新的字节,则维持“1”的状态,使发送数据线处于空闲。从开始位到停止位结束的时间间隔称为一字节帧。串行通信的速度用波特率表征,其含义是把每秒内传送的位数,单位是位/秒,记为b/s,最典型的波特率是9600。

(2) 给出了UART驱动构件有9个对外接口函数:初始化(uart_init)、发送单个字节(uart_send1)、发送N个字节(uart_sendN)、发送字符串(uart_send_string)、接收单个字节(uart_re1)、接收N个字节(uart_reN)、使能串口接收中断(uart_enable_re_int)、禁止串口接收中断(uart_disable_re_int)、获取接收中断状态(uart_get_re_int)。在UART驱动构件的头文件(uart.h)中还用宏定义方式统一了使用的串口号(UART_0、UART_1、UART_2),以及实际使用时它们所在的引脚组。另外还给出串口printf函数,方便嵌入式调试。

(3) 给出了关于中断的通用基础知识、ARM Cortex-M0+非内核模块中断编程结构,以串口接收中断为例,给出了中断编程步骤及范例。网上教学资源中还给出了PC方C#串口测试源程序。掌握一门可以与MCU通信的PC编程语言,并合理地加以应用,对嵌入式系统的学习将有很大帮助。

(4) 给出了 UART 驱动构件的设计方法。这项工作有一定难度，可以根据自己的学习情况确定掌握深度。基本要求是在重点掌握控制寄存器、状态寄存器中加底纹位段基础上，理解初始化、发送一个字节、接收一个字节函数。

(5) 本书网上教学资源中的补充阅读材料给出了 DMA 的简要讨论。

习　　题

1. 简述 MCU 与 PC 之间进行串行通信，为什么要进行电平转换？如何进行电平转换？

2. 设波特率为 9600，使用 NRZ 格式的 8 个数据位、没有校验位、1 个停止位，传输 2KB 的文件最少需要多少时间？

3. 简要阐述 UART 驱动构件的使用方法。

4. 简述 M0＋中断机制及运行过程。

5. 用一种高级语言（例如 C♯）实现 PC 方串行通信数据收发的通用程序。

6. 编写程序实现通过 PC 软件控制与 MCU 相连的三盏指示灯的亮暗状态。（提示：PC 与 MCU 之间通过 UART 通信。）

7. 阐述设计 UART 构件的知识要素。

8. 说明 UART 构件中对引脚复用的处理方法及优缺点。

第 7 章　定时器相关模块

本章导读：本章阐述 KL 系列 MCU 与定时器相关的几个模块的编程。主要内容有：7.1 节介绍内核时钟 SysTick；7.2～7.4 节介绍脉宽调制、输入捕捉与输出比较通用基础知识、TPM 模块的驱动构件及使用方法、定时器/PWM 模块（TPM）编程结构及 TPM 模块驱动构件的设计方法；7.4～7.7 节分别介绍周期性中断定时器（PIT）、低功耗定时器（LPTMR）、实时时钟模块（RTC）的功能、构件及使用方法、构件的设计方法。

本章参考资料：7.1 节（SysTick）参考《M0＋用户指南》4.4 节及《KL 参考手册》3.3.1 节；7.3 和 7.4 节（TPM）参考《KL 参考手册》第 31 章；7.5 节（PIT）参考《KL 参考手册》第 32 章；7.6 节（LPTMR）来自《KL 参考手册》第 33 章；7.7 节（RTC）参考《KL 参考手册》第 34 章。

在嵌入式应用系统中，有时要求能对外部脉冲信号或开关信号进行计数，这可通过计数器来完成。有些设备要求每间隔一定时间开启并在一段时间后关闭，有些指示灯要求不断地闪烁，这可利用定时信号来完成。另外，系统日历时钟、产生不同频率的声源等也需要定时信号。计数与定时问题的解决方法是一致的，只不过是同一个问题的两种表现形式。实现计数与定时的基本方法有三种：完全硬件方式、完全软件方式、可编程计数器/定时器。完全硬件方式基于逻辑电路实现，现已很少使用。完全软件方式是利用计算机执行指令的时间实现定时，但这种方式占用 CPU，不适用于多任务环境，一般仅用于时间极短的延时且重复次数较少的情况。更常用的是可编程定时器，它在设定之后与 CPU 并行地工作，不占用 CPU 的工作时间。这种方法的主要思想是根据需要的定时时间，用指令对定时器设置定时常数，并用指令启动定时器开始计数，当计数到指定值时，便自动产生一个定时输出或中断信号告知 CPU。在定时器开始工作以后，CPU 不必去管它，而可以去做其他工作。如果利用定时器产生中断信号还可以建立多任务环境，可大大提高 CPU 的利用率。本章后续阐述的均是这种类型的定时器。

7.1　ARM Cortex-M0＋内核定时器

ARM Cortex-M 内核中包含一个简单的定时器 SysTick，又称为“滴答”定时器。SysTick 定时器被捆绑在 NVIC（嵌套向量中断控制器）中，有效位数是 24 位，采用减 1 计数的方式工作，当减 1 计数到 0，可产生 SysTick 异常（中断），中断号为 15。

嵌入式操作系统或使用了时基的嵌入式应用系统，都必须由一个硬件定时器来产生需要的“滴答”中断，作为整个系统的时基。由于所有使用 Cortex-M 内核的芯片都带有 SysTick，并且在这些芯片中，SysTick 的处理方式（寄存器映射地址及作用）都是相同的，若使用 SysTick 产生时间“滴答”，可以化简嵌入式软件在 Cortex-M 内核芯片间的移植工作。

7.1.1　SysTick 模块的编程结构

1. SysTick 定时器模块的寄存器地址

SysTick 定时器模块中有 4 个 32 位寄存器，其映像地址及简明功能见表 7-1。

表 7-1　SysTick 模块的寄存器映像地址及简明功能

寄存器名	简　称	访问地址	简明功能
控制及状态寄存器	SYST_CSR	0xE000_E010	配置功能及状态标志
重载寄存器	SYST_RVR	0xE000_E014	低 24 位有效，计数器到 0，用该寄存器的值重载
计数器	SYST_CVR	0xE000_E018	低 24 位有效，计数器当前值，减 1 计数
校准寄存器	SYST_CALIB	0xE000_E01C	针对不同 MCU，校准恒定中断频率。KL25 未用

2. 控制及状态寄存器 SYST_CSR

控制及状态寄存器 SYST_CSR 见表 7-2。主要有溢出标志位 COUNTFLAG、时钟源选择位 CLKSOURCE、中断使能控制位 TICKINT 和 SysTick 模块使能位 ENABLE。复位时，各位为 0。

表 7-2　控制及状态寄存器 SYST_CSR

位	名　称	R/W	功能说明
16	COUNTFLAG	R	计数器减 1 计数到 0，则该位为 1；读取该位清 0
2	CLKSOURCE	R/W	＝0，外部时钟(KL 选此项，为内核时钟/16)；＝1，内核时钟
1	TICKINT	R/W	＝0，禁止中断；＝1，允许中断(计数器到 0 时，中断)
0	ENABLE	R/W	SysTick 模块使能位，＝0，关闭；＝1，使能

3. 计数器 SYST_CVR 及重载寄存器 SYST_RVR

SysTick 模块的计数器 SYST_CVR 保存当前计数值，这个寄存器是由芯片硬件自行维护，用户无须干预，系统可通过读取该寄存器的值得到更精细的时间表示。SysTick 模块的重载寄存器 SYST_RVR 的低 24 位 D23～D0(RELOAD)有效，其值是计数器的初值及重载值。

SysTick 模块内的计数器 SYST_CVR 是一个 24 位计数器，减 1 计数。初始化时，选择时钟源(决定了计数频率)、设置重载寄存器 SYST_RVR、设置优先级、允许中断，计数器的初值为“重载寄存器 SYST_RVR”中的值、使能该模块。则计数器开始减 1 计数，计数到 0 时，SysTick 控制及状态寄存器 SYST_CSR 的溢出标志位 COUNTFLAG 被置 1，产生中断请求，同时，计数器自动重载初值并继续减 1 计数。

4. M0＋内核优先级设置寄存器

编写 SysTick 模块的初始化程序还需用到 M0＋内核优先级设置寄存器(System Handler Priority Register 3，SHPR3)，用于设定 SysTick 模块中断的优先级。SHPR3 位于系统控制块(System Control Block，SCB)中。在 ARM Cortex-M0＋中，只有 SysTick、SVC(系统服务调用)和 PendSV(可挂起系统调用)等内部异常可以设置其中断优先级，其他内核异常的优先级是固定的。SVC 的优先级在 SHPR2 寄存器中设置，SysTick 和 PendSV 优先级在 SHPR3 寄存器中设置，见图 7-1。

bit	31 30	24 23 22	16 15 14	8 7 6	0	
0xE000ED20	SysTick		PendSV			SHPR3
0xE000ED1C	SVC					SHPR2

图 7-1 SysTick 优先级寄存器

SHPR3 寄存器的最高两位 D31、D30 设置 SysTick 优先级，优先级可以设为 0～3 级，一般设为 3。

7.1.2 SysTick 构件设计及测试工程

下面以 SysTick 定时器模块为时钟源，每隔 1s 通过串口向 PC 发送时、分和秒。

1. SysTick 构件头文件(systick.h)

```
//=========================================================================
//文件名称：systick.h
//功能概要：systick 定时器模块构件头文件
//版权所有：苏州大学恩智浦嵌入式中心(sumcu.suda.edu.cn)
//更新记录：2016-03-20    V4.0
//=========================================================================

#ifndef _SYSTICK_H
#define _SYSTICK_H

#include "common.h"
//#include "sysinit.h"


//时钟源宏定义
#define CORE_CLOCK                 1
#define CORE_CLOCK_DIV_16          0
//=========================================================================
//函数名称：systick_init
//函数返回：无
//参数说明：clk_src_sel：时钟源选择：1：内核时钟(core_clk_khz)；
//                                  0：内核时钟/16。
//          int_ms:中断的时间间隔。单位 ms 推荐选用 5,10,…,最大为 50
//功能概要：初始化 SysTick 模块，设置中断的时间间隔
//说明：内核时钟频率 core_clk_khz 宏定义在 sysinit.h 中
//=========================================================================
void systick_init(uint_8 clk_src_sel, uint_8 int_ms);

#endif
```

2. SysTick 构件源文件(systick.c)

```
//=========================================================================
//文件名称：systick.c
//功能概要：systick 定时器模块构件源文件
```

```
//版权所有：苏州大学恩智浦嵌入式中心(sumcu.suda.edu.cn)
//更新记录：2016-03-20   V4.0
//==========================================================
#include "systick.h"

//==========================================================
//函数名称：systick_init
//函数返回：无
//参数说明：clk_src_sel：时钟源选择：1：内核时钟(CORE_CLK_KHZ)；
//                                  0：内核时钟/16。
//          int_ms:中断的时间间隔。单位 ms 推荐选用 5,10,…,最大为 50
//功能概要：初始化 SysTick 模块,设置中断的时间间隔
//说    明：内核时钟频率 SYSTEM_CLK_KHZ 宏定义在 common.h 中
//          systick 以 ms 为单位,最大可为 349(2^24/48000,向下取整),合理范围 1～349。
//          前提是时钟是内核时钟,为 48MHz。假如时钟频率升高,合理范围会缩小。
//          KL25 的 SysTick 时钟源可以是内核时钟,也可设置为内核时钟的 16 分频。
//          24 位计数器,减 1 计数
//时间范围：1～349ms(内核时钟);1～5592ms(内核时钟的 16 分频)
//==========================================================
void systick_init(uint_8 clk_src_sel, uint_8 int_ms)
{
    SysTick->CTRL = 0;              //设置前先关闭 systick
    SysTick->VAL = 0;               //清除计数器

    //根据计数频率,确定并设置重载寄存器的值
    if(0==clk_src_sel)              //0:内核时钟/16
    {
        if((int_ms<1)&&(int_ms>5592))
        {
            int_ms = 10;
        }
        SysTick->LOAD = SYSTEM_CLK_KHZ * int_ms/16;
    }
    else                            //1:内核时钟
    {
        if((int_ms<1)&&(int_ms>349))
        {
            int_ms = 10;
        }
        SysTick->LOAD = SYSTEM_CLK_KHZ * int_ms;
        SysTick->CTRL = (SysTick_CTRL_CLKSOURCE_Msk);
    }
    //设定 SysTick 优先级为 3(SHPR3 寄存器的最高字节=0xC0)
    NVIC_SetPriority(SysTick_IRQn, (1UL << __NVIC_PRIO_BITS)-1UL);
    //设置时钟源,允许中断,使能该模块,开始计数
    SysTick->CTRL |= (SysTick_CTRL_ENABLE_Msk|SysTick_CTRL_TICKINT_Msk);
}
```

3. SysTick 构件中断服务子程序

```
//=================中断函数服务例程=================
//===============================================
//函数名称: SysTick_Handler
//参数说明: 无
//函数返回: 无
//功能概要: SysTick 定时器中断服务例程
//===============================================
void SysTick_Handler(void)
{
    static uint_8 SysTickcount = 0;
    SysTickcount++;
    if(SysTickcount >= 100)            //1s 到
    {
        SysTickcount = 0;
        //秒计时程序
        SecAdd1(g_time);               //g_time 是时分秒全局变量数组
    }
}
```

4. SysTick 构件测试工程

测试工程位于网上教学资源中的"..\KL25_Systick"文件夹,其功能概述如下。

(1) 串口通信格式:波特率 9600,1 位停止位,无校验。

(2) SysTick 使用内核时钟的 16 分频,即工作时钟为 3MHz。

(3) 上电或按复位按钮时,调试串口输出"苏州大学嵌入式实验室 SysTick 构件测试用例!"。

(4) SysTick 每 10ms 中断一次,在中断里再进行计数判断,每 100 个 SysTick 中断 LIGHT_BLUE 灯状态改变,同时调试串口输出"MCU 记录的相对时间: 00:00:01"。"00:00:01"为 SysTick 记录的时间。

(5) PC 向 MCU 发送数据时,MCU 进入串口接收中断,串口 1 将接收的一个字节直接回发。

7.2　脉宽调制、输入捕捉与输出比较通用基础知识

7.2.1　脉宽调制 PWM 通用基础知识

1. PWM 的基本概念与技术指标

脉宽调制(Pulse Width Modulator,PWM)是电机控制的重要方式之一。PWM 信号是一个高/低电平重复交替的输出信号,通常也叫脉宽调制波或 PWM 波。图 7-2 就给出了 PWM 波的实例。通过 MCU 输出 PWM 信号的方法与使用纯电力电子实现的方法相比,有操作简单、实现方便等优点,所以目前经常使用的 PWM 信号主要通过配置 MCU 的方法实现。这个方法需要有个产生 PWM 波的时钟源,设其周期为 T_{CLK}。PWM 信号的主要技术

指标有周期、占空比、极性、脉冲宽度、分辨率、对齐方式等。下面分别介绍。

1）PWM 周期

PWM 信号的周期用其持续的时钟周期个数来度量。例如，图 7-2 中的 PWM 信号的周期是 8 个时钟周期，即 $T_{PWM}=8T_{CLK}$。

2）PWM 占空比

PWM 占空比被定义为 PWM 信号处于有效电平的时钟周期数与整个 PWM 周期内的时钟周期数之比，用百分比表征。图 7-2（a）中，PWM 的高电平（高电平为有效电平）为 $2T_{CLK}$，所以占空比＝2/8＝25％，类似计算，图 7-2（b）占空比为 50％（方波）、图 7-2（c）占空比为 75％。

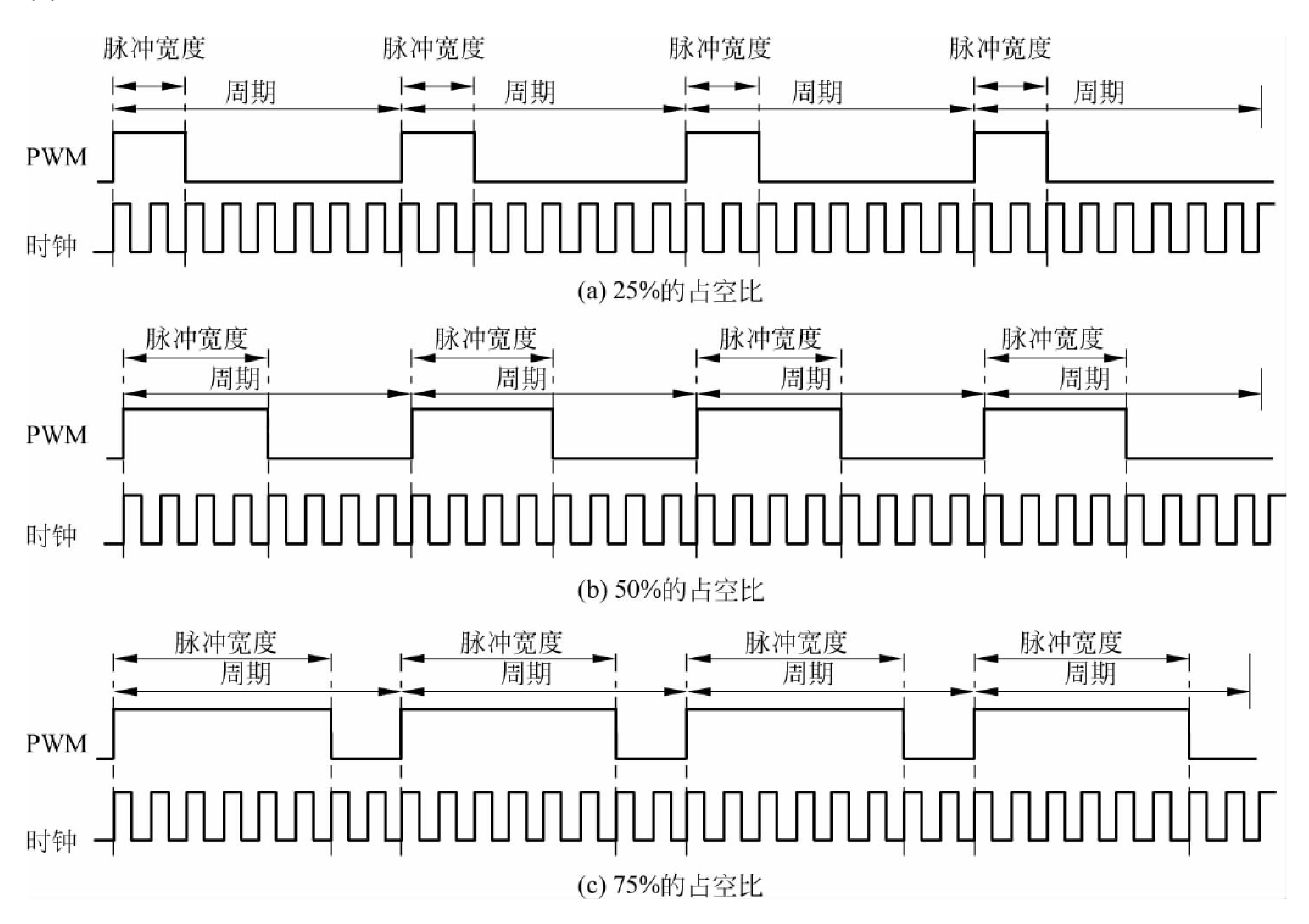


图 7-2　PWM 的占空比的计算方法

3）PWM 极性

PWM 极性决定了 PWM 波的有效电平。正极性，表示 PWM 有效电平为高（图 7-2），那么在边沿对齐的情况下，PWM 引脚的平时电平（也称空闲电平）就应该为低，开始产生 PWM 的信号为高电平，到达比较值时，跳变为低电平，到达 PWM 周期时又变为高电平，周而复始。负极性则相反，PWM 引脚平时电平（空闲电平）为高，有效电平为低。但注意，占空比通常仍定义为高电平时间与 PWM 周期之比。

4）脉冲宽度

脉冲宽度是指一个 PWM 周期内，PWM 波处于高电平的时间（用持续的时钟周期数表征）。可以用占空比与周期计算出来，可不作为一个独立的技术指标。

5）PWM 分辨率

PWM 分辨率 ΔT 是指脉冲宽度的最小时间增量。例如，若 PWM 是利用频率为

48MHz 的时钟源产生的，即时钟源周期==(1/48)μs=0.208μs=20.8ns，那么脉冲宽度的每一增量为 $\Delta T=20.8\text{ns}$，就是 PWM 的分辨率。它就是脉冲宽度的最小时间增量了，脉冲宽度的增加与减少只能是 ΔT 的整数倍。实际上，脉冲宽度正是用高电平持续的时钟周期数(整数)来表征。

6) PWM 的对齐方式

可以用 PWM 引脚输出发生跳变的时刻来描述 PWM 的边沿对齐与中心对齐两种对齐方式。从 MCU 编程方式产生 PWM 的方法来理解。设产生 PWM 波的时钟源周期为 T_{CLK}，PWM 的周期 $T_{PWM}==M\times T_{CLK}$，脉宽 $W=N\times T_{CLK}$，同时假设 $N>0$，$N<M$，计数器记为 CNT，通道(n)值寄存器记为 CnV=N，用于比较。设 PWM 引脚输出平时电平为低电平，开始时，CNT 从 0 开始计数，在 CNT=0 的时钟信号上升沿，PWM 输出引脚由低变高，随着时钟信号增 1，CNT 增 1，当 CNT=N 时(即 CNT=CnV)，在此刻的时钟信号上升沿，PWM 输出引脚由高变低，持续 $M-N$ 个时钟周期，CNT=0，PWM 输出引脚由低变高，周而复始。这就是边沿对齐(Edge-Aligned)的 PWM 波，缩写为 EPWM，是一种常用 PWM 波。图 7-3 给出了周期为 8，占空比为 25%的 EPWM 波示意图。可以概括地说，在平时电平为低电平的 PWM 的情况下，开始计数时，PWM 引脚同步变高，就是边沿对齐。

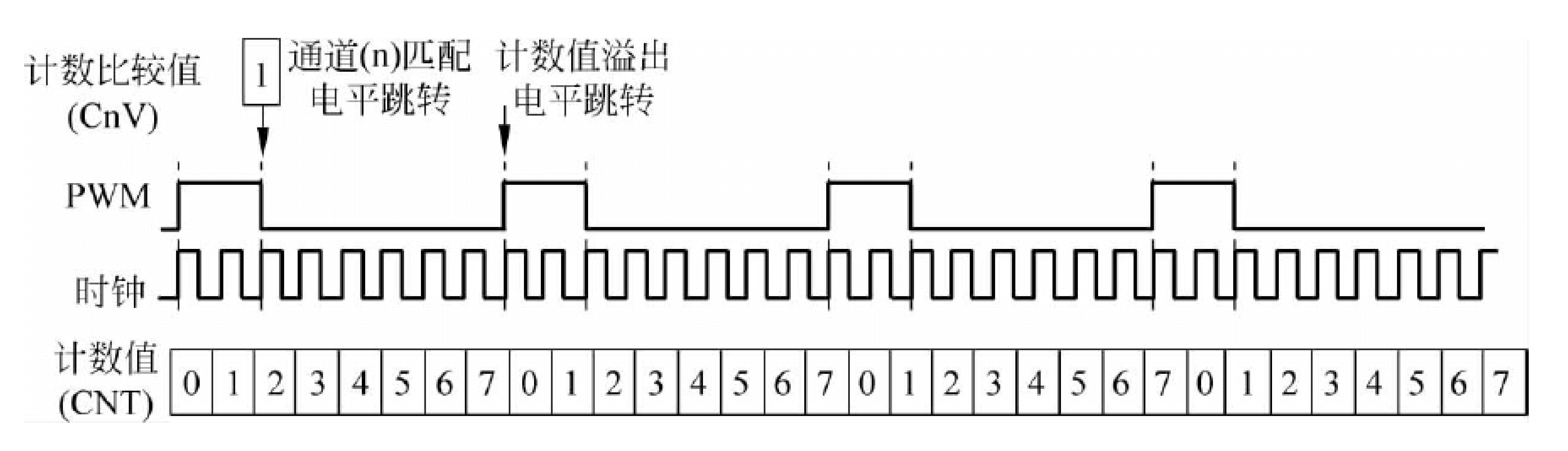


图 7-3 边沿对齐方式 PWM 输出

中心对齐 PWM(Center-Aligned)的 PWM 波，缩写为 CPWM，是一种比较特殊的产生 PWM 脉宽调制波的方法，常用在逆变器、电机控制等场合。图 7-4 给出了 25%占空比时 CPWM 产生的示意图，在计数器向上计数时，当计数值(CNT)小于计数比较值(CnV)的时候，PWM 通道输出低电平，当计数值(CNT)大于计数比较值(CnV)的时候，PWM 通道发生电平跳转输出高电平。在计数器向下计数时，当计数值(CNT)大于计数比较值(CnV)的时候，PWM 通道输出高电平，当计数值(CNT)小于计数比较值(CnV)的时候，PWM 通道发生电平跳转输出低电平。按此运行机理周而复始地运行便实现 CPWM 波的正常输出。可以概括地说，设 PWM 波的低电平时间 $t_L=K\times T_{CLK}$，在平时电平为低电平的 PWM 的情况下，中心对齐的 PWM 波，比边沿对齐的 PWM 波形向右平移了($K/2$)个时钟周期。

本书网上教学资源中的补充阅读材料给出了边沿对齐和中心对齐方式应用场景简介。

2. PWM 的应用场合

PWM 最常见的应用是电机控制。还有一些其他用途，这里仅举几例。①利用 PWM 为其他设备产生类似于时钟的信号。例如，PWM 可用来控制灯以一定频率闪烁。②利用 PWM 控制输入到某个设备的平均电流或电压。例如，一个直流电机在输入电压时会转动，而转速与平均输入电压的大小成正比。假设每分钟转速(rpm)=输入电压的 100 倍，如果转速要达到 125rpm，则需要 1.25V 的平均输入电压；如果转速要达到 250rpm，则需要

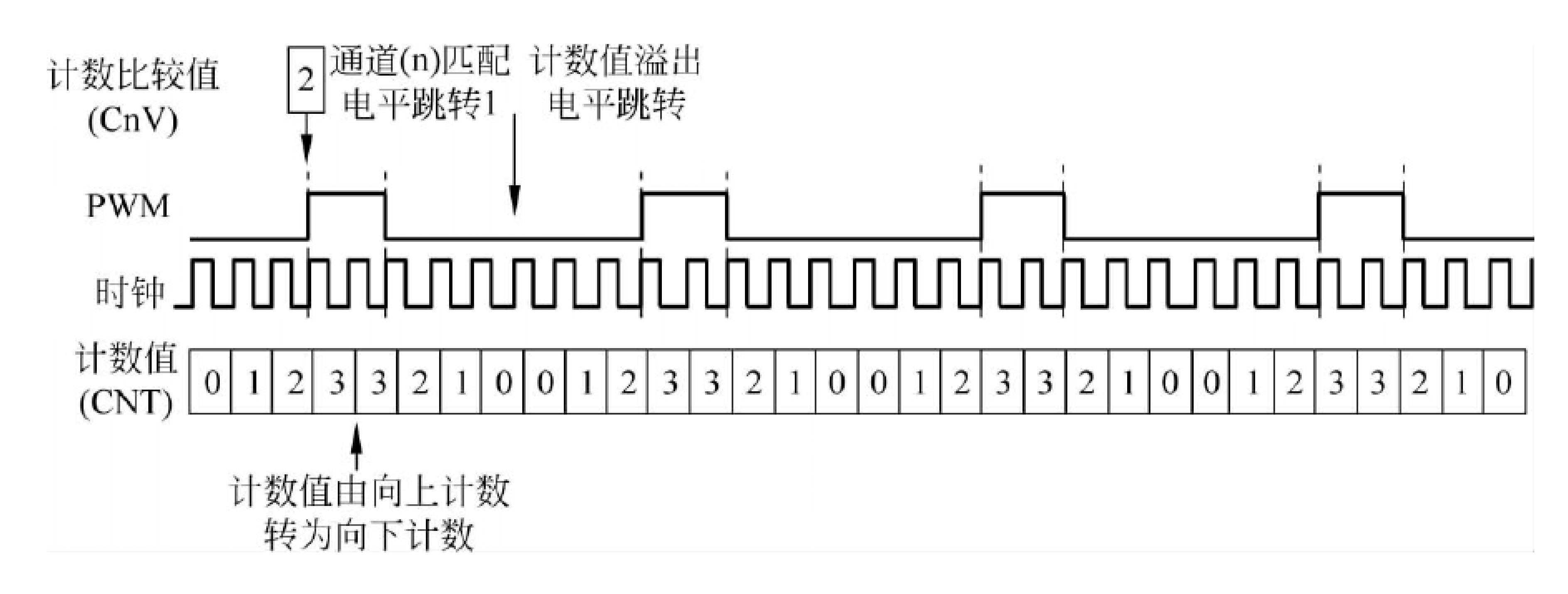


图 7-4　25%占空比中心对齐方式 PWM

2.50V 的平均输入电压。在图 7-2 中，如果逻辑 1 是 5V，逻辑 0 是 0V，则图 7-2(a)的平均电压是 1.25V，图 7-2(b)的平均电压是 2.5V，图 7-2(c)的平均电压是 3.75V。可见，利用 PWM 可以设置适当的占空比来得到所需的平均电压，如果所设置的周期足够小，电机就可以平稳运转(即不会明显感觉到电机在加速或减速)。③利用 PWM 控制命令字编码。例如，通过发送不同宽度的脉冲，代表不同含义。假如用此来控制无线遥控车，宽度 1ms 代表左转命令，4ms 代表右转命令，8ms 代表前进命令。接收端可以使用定时器来测量脉冲宽度，在脉冲开始时启动定时器，脉冲结束时停止定时器，由此来确定所经过的时间，从而判断收到的命令。

7.2.2　输入捕捉与输出比较通用基础知识

1. 输入捕捉的基本含义与应用场合

输入捕捉是用来监测外部开关量输入信号变化的时刻。当外部信号在指定的 MCU 输入捕捉引脚上发生一个沿跳变(上升沿或下降沿)时，定时器捕捉到沿跳变之后，把计数器当前值锁存到通道寄存器，同时产生输入捕捉中断，利用中断处理程序可以得到沿跳变的时刻。这个时刻是定时器工作基础上的更精细时刻。

输入捕捉的应用场合主要有测量脉冲信号的周期与波形。例如，自己编程产生的 PWM 波，可以直接连接输入捕捉引脚，通过输入捕捉的方法测量回来，看看是否达到要求。输入捕捉的应用场合还有电机的速度测量。本书网上教学资源中的补充阅读材料利用输入捕捉测量电机速度方法简介。

2. 输出比较的基本含义与应用场合

输出比较的功能是用程序的方法在规定的较精确时刻输出需要的电平，实现对外部电路的控制。MCU 输出比较模块的基本工作原理是，当定时器的某一通道用作输出比较功能时，通道寄存器的值(CnV)和计数寄存器(CNT)的值每隔 4 个总线周期比较一次。当两个值相等时，输出比较模块置定时器通道状态和控制寄存器(CnSC)的通道标志 CHF 位为 1，并且在该通道的引脚上输出预先规定的电平。如果输出比较中断允许，还会产生一个中断。

输出比较的应用场合主要有产生一定间隔的脉冲，典型的应用实例就是实现软件的串行通信。用输入捕捉作为数据输入，而用输出比较作为数据输出。首先根据通信的波特率向通道寄存器写入延时的值，根据待传的数据位确定有效输出电平的高低。在输出比较中

断处理程序中,重新更改通道寄存器的值,并根据下一位数据改写有效输出电平控制位。

7.3　TPM模块的驱动构件及使用方法

定时器/脉宽调制模块(Timer/PWM Module,TPM)内含三个模块,分别称为TPM0、TPM1、TPM2,每个模块是独立的。TPM模块,除了作为基本定时器外,主要用于支持PWM、输入捕捉、输出比较功能。TPM0有6个通道,TPM1和TPM2只有两个通道。每个定时器有个中断向量,由表3-6,TPM0～2中断向量号分别为33～35,IRQ中断号分别为17～19。在"startup_MKL25Z4.s"文件中设定的默认中断处理程序名分别为TPM0_IRQHandler、TPM1_IRQHandler、TPM2_IRQHandler,在工程框架的"isr.c"实现中断处理程序。

每个TPM模块均具有16位计数器(CNT)、模数寄存器(MOD)、状态可控制寄存器(SC)、第n通道的通道值寄存器(CnV)及通道的状态可控制寄存器(CnSC)、输入捕捉和输出比较状态寄存器(STATUS)、配置寄存器(CONF)。TPM模块虽然具有基本计时功能,但由于KL系列具有诸多具有计时功能的定时器,如7.1节给出的内核的SysTick及7.5～7.7节分别给出的周期中断定时器PIT、低功耗定时器LPTMR、实时时钟RTC,因此,TPM模块一般作为脉宽调制、输入捕捉、输出比较功能使用。

7.3.1　TPM模块的脉宽调制、输入捕捉、输出比较引脚

表7-3列出了TPM模块用于脉宽调制、输入捕捉、输出比较功能的外部引脚。

表7-3　KL25的TPM的外部引脚复用功能

引脚号	引脚名	ALT0	ALT1	ALT2	ALT3	ALT4
13	PTE20	ADC0_DP0/ADC0_SE0	PTE20		TPM1_CH0	UART0_TX
14	PTE21	ADC0_DM0/ADC0_SE4a	PTE21		TPM1_CH1	UART0_RX
15	PTE22	ADC0_DP3/ADC0_SE3	PTE22		TPM2_CH0	UART2_TX
16	PTE23	ADC0_DM3/ADC0_SE7a	PTE23		TPM2_CH1	UART2_RX
21	PTE29	CMP0_IN5/ADC0_SE4b	PTE29		TPM0_CH2	TPM_CLKIN0
22	PTE30	DAC0_OUT/ADC0_SE23/CMP0_IN4	PTE30		TPM0_CH3	TPM_CLKIN1
23	PTE31		PTE31		TPM0_CH4	
24	PTE24		PTE24		TPM0_CH0	
25	PTE25		PTE25		TPM0_CH1	
26	PTA0	TSI0_CH1	PTA0		TPM0_CH5	
27	PTA1	TSI0_CH2	PTA1	UART0_RX	TPM2_CH0	
28	PTA2	TSI0_CH3	PTA2	UART0_TX	TPM2_CH1	
29	PTA3	TSI0_CH4	PTA3	I2C1_SCL	TPM0_CH0	
30	PTA4	TSI0_CH5	PTA4	I2C1_SDA	TPM0_CH1	
31	PTA5		PTA5	USB_CLKIN	TPM0_CH2	
32	PTA12		PTA12		TPM1_CH0	

续表

引脚号	引脚名	ALT0	ALT1	ALT2	ALT3	ALT4
33	PTA13		PTA13		TPM1_CH1	
40	PTA18	EXTAL0	PTA18		UART1_RX	TPM_CLKIN0
41	PTA19	XTAL0	PTA19		UART1_TX	TPM_CLKIN1
43	PTB0	ADC0_SE8/TSI0_CH0	PTB0/LLWU_P5	I2C0_SCL	TPM1_CH0	
44	PTB1	ADC0_SE9/TSI0_CH6	PTB1	I2C0_SDA	TPM1_CH1	
45	PTB2	ADC0_SE12/TSI0_CH7	PTB2	I2C0_SCL	TPM2_CH0	
46	PTB3	ADC0_SE13/TSI0_CH8	PTB3	I2C0_SDA	TPM2_CH1	
51	PTB16	TSI0_CH9	PTB16	SPI1_MOSI	UART0_RX	TPM_CLKIN0
52	PTB17	TSI0_CH10	PTB17	SPI1_MISO	UART0_TX	TPM_CLKIN1
53	PTB18	TSI0_CH11	PTB18		TPM2_CH0	
54	PTB19	TSI0_CH12	PTB19		TPM2_CH1	
56	PTC1	ADC0_SE15	PTC1/	I2C1_SCL		TPM0_CH0
57	PTC2	ADC0_SE11/TSI0_CH15	PTC2	I2C1_SDA		TPM0_CH1
58	PTC3		PTC3/LLWU_P7		UART1_RX	TPM0_CH2
61	PTC4		PTC4/LLWU_P8	SPI0_PCS0	UART1_TX	TPM0_CH3
65	PTC8	CMP0_IN2	PTC8	I2C0_SCL	TPM0_CH4	
66	PTC9	CMP0_IN3	PTC9	I2C0_SDA	TPM0_CH5	
69	PTC12		PTC12			TPM_CLKIN0
70	PTC13		PTC13			TPM_CLKIN1
73	PTD0		PTD0	SPI0_PCS0		TPM0_CH0
74	PTD1	ADC0_SE5b	PTD1	SPI0_SCK		TPM0_CH1
75	PTD2		PTD2	SPI0_MOSI	UART2_RX	TPM0_CH2
76	PTD3		PTD3	SPI0_MISO	UART2_TX	TPM0_CH3
77	PTD4		PTD4/LLWU_P14	SPI1_PCS0	UART2_RX	TPM0_CH4
78	PTD5	ADC0_SE6b	PTD5	SPI1_SCK	UART2_TX	TPM0_CH5

7.3.2　TPM 构件头文件

TPM 驱动构件由头文件 tpm.h 及源代码文件 tpm.c 组成，放入 tpm 文件夹中，供应用程序开发调用。

TPM 模块通常用作输入捕捉、输出比较或 PWM 输出三种基本功能。下面分析 TPM 初始化函数都需要哪些参数。首先应该有 TPM 模块号和通道号，因为当使用 TPM 的上述三个功能时必须把相应功能映射到不同的 TPM 通道上；其次是定时器计数溢出值，通过设定这个值来确定 TPM 定时器的基本周期，因为必须先确定 TPM 定时器的基本定时周期才可以对占空比、对齐方式等参数进行设定。至于周期的大小，则由总线时钟和模块时钟的分频器来决定，分频系数作为参数可以传入 TPM 时钟初始化函数。这样，TPM 初始化函数就有两个参数：TPM 模块号和通道号与 TPM 计数器周期。KL 系列 MCU 的一组 TPM 通道，可以在不同引脚组上，实际应用中使用哪个引脚，应该是在应用开发板硬件设计阶段就确定的，为了使驱动构件适应这个场景，可在头文件中使用“宏”进行定义，确定 TPM 通道使用的引脚。这个方法也有缺点，就是若把源代码文件编译成库，再修改宏定义就不起作

用了,必须重新使用源程序进行编译,这是所有宏定义的共性。

从知识要素角度,进一步分析TPM驱动构件的基本函数,完成了TPM初始化的函数后若要实现输入捕捉、输出比较或PWM输出三种基本功能,还需要对其进行不同的功能初始化配置。其中,输入捕捉初始化函数添加了输入捕捉模式选择一个参数,输出比较初始化函数则加入了占空比和输出比较模式选择两个参数,PWM信号输出对应地加入了占空比、对齐方式、极性三个可修改参数。

TPM模块计数器位数为16位,计数范围为0~65 535,计数方式采取了向上计数的方式,TPM时钟为48MHz,最小定时时间为20.8ns,最大定时时间为1.36ms。TPM中断服务例程名为TPM0_IRQHandler,进入TPM中断后通过清除TOF位来清除中断标志位。

对TPM进行编程,实际上已经涉及对硬件底层寄存器的直接操作,因此可以将中断使能、初始化、关闭等基本操作所对应的功能函数共同定义在命名为tpm.c的文件中,并按照相对严格的构件设计原则对其进行封装,同时配以命名为tpm.h的头文件,用来定义模块的基本信息和对外接口。将中断函数定义在名为isr.c的文件中。下面通过封装TPM基本功能函数来进一步深刻理解构件化的编程思想。

```
//=========================================================================
//文件名称：tpm.h
//功能概要：tpm底层驱动构件源文件
//版权所有：苏州大学NXP嵌入式中心(sumcu.suda.edu.cn)
//更新记录：2013-6-20   V1.0,2013-6-10   V3.0
//备注：SD-FSL-KL25开发板有TPM0、TPM1和TPM2共3个TPM模块,每个模块又有
//            若干通道,都可以配置产生边沿对齐或是中心对齐的PWM信号。
//前提：PTB9、PTB18、PTB19已分配给三色灯,PTE0、PTE1分配给UART1,PTE22、PTE23分配
//给UART2
//PTA0、PTA3、PTA20分配给调试接口使用
//            每个通道的具体引脚分配如下：
//              通道                   引脚名(复用编号)
//              TPM0_CH0       [PTA3(3)]、PTC1(4)、PTD0(4)、PTE24(3)
//              TPM0_CH1       PTA4(3)、PTC2(4)、PTD1(4)、PTE25(3)
//              TPM0_CH2       PTA5(3)、PTC3(4)、PTD2(4)、PTE29(3)
//              TPM0_CH3       PTC4(4)、PTD3(4) 、PTE30(3)
//              TPM0_CH4       PTC8(3)、PTD4(4)、PTE31(3)
//              TPM0_CH5       [PTA0(3)]、PTC9(3)、PTD5(4)
//
//              TPM1_CH0       PTA12(3)、PTB0(3)、PTE20(3)
//              TPM1_CH1       PTA13(3)、PTB1(3)、PTE21(3)
//
//              TPM2_CH0       PTA1(3)、PTB2(3)、[PTB18(3)]、[PTE22(3)]
//              TPM2_CH1       PTA2(3)、PTB3(3)、[PTB19(3)]、[PTE23(3)]
//=========================================================================
#ifndef _TPM_H
#define _TPM_H

#include "common.h"
#include "gpio.h"
//TPM模块号宏定义
#define TPM_0      0
#define TPM_1      1
```

```
#define TPM_2      2

//输入捕捉边沿获取模式宏定义
#define CAP_UP       0
#define CAP_DOWN      1
#define CAP_DOUBLE   2
//输出比较模式选择宏定义
#define CMP_REV    0
#define CMP_LOW    1
#define CMP_HIGH   2
//PWM 对齐方式宏定义:边沿对齐、中心对齐
#define PWM_EDGE    0
#define PWM_CENTER 1
//PWM 极性选择宏定义: 正极性、负极性
#define PWM_PLUS    0
#define PWM_MINUS   1

//注:通过展开以下宏定义修改宏定义值可选择多引脚通道的一个引脚
//-------------------TPM0 通道的引脚选择--------------------
//TPM0 通道 0 的引脚:
(PTA_NUM|3),(PTC_NUM|1),(PTD_NUM|0),(PTE_NUM|24)
#define TPM0_CH0 (PTC_NUM|1)
//TPM0 通道 1 的引脚:
(PTA_NUM|4),(PTC_NUM|2),(PTD_NUM|1),(PTE_NUM|25)
#define TPM0_CH1 (PTA_NUM|4)
//TPM0 通道 2 的引脚:
(PTA_NUM|5),(PTC_NUM|3),(PTD_NUM|2),(PTE_NUM|29)
#define TPM0_CH2 (PTA_NUM|5)
//TPM0 通道 3 的引脚:
(PTC_NUM|4),(PTD_NUM|3),(PTE_NUM|30)
#define TPM0_CH3 (PTC_NUM|4)
//TPM0 通道 4 的引脚:
(PTC_NUM|8),(PTD_NUM|4),(PTE_NUM|31)
#define TPM0_CH4 (PTC_NUM|8)
//TPM0 通道 5 的引脚:
(PTA_NUM|0),(PTC_NUM|9),(PTD_NUM|5)
#define TPM0_CH5 (PTA_NUM|0)
//---------------------------------------------------------

//--------------TPM1 通道的引脚选择---------------------
//TPM1 通道 0 的引脚:
(PTA_NUM|12),(PTB_NUM|0),(PTE_NUM|20)
#define TPM1_CH0 (PTA_NUM|12)
//TPM1 通道 1 的引脚:
(PTA_NUM|13),(PTB_NUM|1),(PTE_NUM|21)
#define TPM1_CH1 (PTA_NUM|13)
//----------------------------------------------------

//-----------------TPM2 通道的引脚选择------------------
//TPM2 通道 0 的引脚:
(PTA_NUM|1),(PTB_NUM|2)
#define TPM2_CH0 (PTA_NUM|1)
//TPM2 通道 1 的引脚:
```

```
(PTA_NUM|2),(PTB_NUM|3)
#define TPM2_CH1 (PTA_NUM|2)
//------------------------------------------------------
//============================================================
//函数名称：tpm_timer_init
//功能概要：tpm模块初始化,设置计数器频率f及计数器溢出时间MOD_Value。
//参数说明：TPM_i：模块号,使用宏定义：TPM_0、TPM_1、TPM_2
//         f：单位：kHz,取值：375、750、1500、3000、6000、12000、24000、48000
//         MOD_Value：单位ms,范围取决于计数器频率与计数器位数(16位)
//函数返回：无
//============================================================
void tpm_timer_init(uint_16 TPM_i,uint32_t f,float MOD_Value);

//============================================================
//函数名称：pwm_init
//功能概要：pwm模块初始化
//参数说明：tpmx_Chy：模块通道号(例：TPM_CH0表示为TPM0模块第0通道)
//         duty：占空比：0.0～100.0对应0%～100%
//           Align：PWM计数对齐方式(有宏定义常数可用)
//         pol：PWM极性选择(有宏定义常数可用)
//函数返回：无
//============================================================
void pwm_init(uint_16 tpmx_Chy,float duty,uint_8 Align,uint_8 pol);

//============================================================
//函数名称：pwm_update
//功能概要：tpmx模块Chy通道的PWM更新
//参数说明：tpmx_Chy：模块通道号(例：TPM_CH0表示为TPM0模块0通道)
//         duty：占空比：0.0～100.0对应0%～100%
//函数返回：无
//============================================================
void pwm_update(uint_16 tpmx_Chy,float duty);

//============================================================
//函数名称：incap_init
//功能概要：incap模块初始化
//参数说明：tpmx_Chy：模块通道号(例：TPM_CH0表示为TPM0模块第0通道)
//         capmode：输入捕捉模式(上升沿、下降沿、双边沿),有宏定义常数使用
//函数返回：无
//============================================================
void incap_init(uint_16 tpmx_Chy,uint_8 capmode);

//============================================================
//函数名称：tpm_get_capvalue
//功能概要：获取tpmx模块Chy通道的计数器当前值
//参数说明：tpmx_Chy：模块通道号(例：TPM_CH0表示为TPM0模块第0通道)
//函数返回：tpmx模块Chy通道的计数器当前值
//============================================================
uint_16 tpm_get_capvalue(uint_16 tpmx_Chy);

//============================================================
//函数名称：outcompare_init
//功能概要：outcompare模块初始化
```

```
//参数说明：tpmx_Chy：模块通道号(例：TPM_CH0 表示为 TPM0 模块第 0 通道)
//          comduty：输出比较电平翻转位置占总周期的比例：0.0～100.0 对应 0%～100%
//          cmpmode：输出比较模式(翻转电平、强制低电平、强制高电平)，有宏定义常数使用
//函数返回：无
//===========================================================================
void outcompare_init(uint_16 tpmx_Chy,float comduty,uint_8 cmpmode);

//===========================================================================
//函数名称：tpm_enable_int
//功能概要：使能 tpm 模块中断
//参数说明：TPM_i：模块号，使用宏定义：TPM_0、TPM_1、TPM_2
//函数返回：无
//===========================================================================
void tpm_enable_int(uint_8 TPM_i);

//===========================================================================
//函数名称：tpm_disable_int
//功能概要：禁用 tpm 模块中断
//参数说明：TPM_i：模块号，使用宏定义：TPM_0、TPM_1、TPM_2
//函数返回：无
//===========================================================================
void tpm_disable_int(uint_8 TPM_i);

//===========================================================================
//函数名称：tpm_get_int
//功能概要：获取 TPM 模块中断标志
//参数说明：TPM_i：模块号，使用宏定义：TPM_0、TPM_1、TPM_2
//函数返回：中断标志 1=有中断产生;0=无中断产生
//===========================================================================
uint_8 tpm_get_int(uint_8 TPM_i);

//===========================================================================
//函数名称：tpm_chl_get_int
//功能概要：获取 TPM 通道中断标志
//参数说明：TPM_i：模块号，使用宏定义：TPM_0、TPM_1、TPM_2，TPMC_i：0～5 通道
//函数返回：中断标志 1=有中断产生;0=无中断产生
//===========================================================================
uint_8 tpm_chl_get_int(uint_8 TPM_i,uint_8 TPMC_i);

//===========================================================================
//函数名称：tpm_clear_int
//功能概要：清除 TPM 中断标志
//参数说明：TPM_i：模块号，使用宏定义：TPM_0、TPM_1、TPM_2
//函数返回：清除中断标志位
//===========================================================================
void tpm_clear_int(uint_8 TPM_i);

//===========================================================================
//函数名称：tpm_clear_chl_int
//功能概要：清除 TPM 通道中断标志
```

```
//参数说明: TPM_i: 模块号,使用宏定义: TPM_0、TPM_1、TPM_2,TPMC_i: 0~5通道
//函数返回: 清除TPM通道中断标志位
//===========================================================
void tpm_clear_chl_int(uint_8 TPM_i,uint_8 TPMC_i);

#endif

//===========================================================
//声明:
//(1)我们开发的源代码,在本中心提供的硬件系统测试通过,真诚奉献给社会,不足之处,
//      欢迎指正。
//(2)对于使用非本中心硬件系统的用户,移植代码时,请仔细根据自己的硬件匹配。
//
//苏州大学嵌入式中心　0512-65214835　http://sumcu.suda.edu.cn
```

7.3.3　TPM 测试工程

TPM 构件 PWM、输入捕捉、输出比较功能的测试工程,在网上教学资源中的"..\program\CH07-KL25-TPM_InCapture-OutCompare-PWM"文件夹中。

1. 测试工程功能概述

(1) 串口通信格式:波特率 9600,1 位停止位,无校验。

(2) 上电或按复位按钮时,调试串口输出"苏州大学嵌入式实验室 TPM-incap-outcomp 构件测试用例!"。

(3) TPM0 基本定时中断,每 10ms 产生一次中断,每中断 100 次累加计时,调用"SecAdd1",进行秒加 1 计时,完成对全局数组 g_time"十分秒"的更新。主程序的无限循环中,判断是否有秒的变化,若变化则通过调试串口输出"时间:时:分:秒",同时蓝色指示灯切换亮暗状态。

(4) 在 TPM1 中断服务例程中,改变 TPM1 模块通道 0 占空比,使其占空比从 0.0 逐渐变大到 100.0,再从 100.0 逐渐变小到 0.0,如此反复,因此可以通过示波器看到 PTA4 输出的方波占空比的变化,从小到大再从大到小,循环执行。

(5) 将 TPM0 的第 0 通道 PTC1 引脚配置为输出比较功能,TPM2 的第 0 通道 PTA1 引脚配置为输入捕捉功能,通过调试串口输出捕捉值。

2. 测试工程的编程步骤

1) 先导工作——在 tpm.h 文件中设定使用的引脚

在工程框架的"..\05_Driver\tpm\tpm.h"文件中,通过宏定义 TPMx_CHy 确定 TPM 模块通道实际使用的引脚,例如,若 TPM0_CH0 实际使用的引脚分别是 PTC1,则做如下设置:

```
//TPM0 通道 0 引脚:
#define TPM0_CH0    (PORT_C|1)     //(PORTA|3),(PORTC|1),(PORTD|0),(PORTE|24)
```

2) main.c 文件中的工作——初始化、使能模块中断、开总中断

在工程框架的"..\ 08_Source\main.c"文件中进行如下编程。

在"初始化外设模块"位置调用 TPM 构件中的初始化函数：

```
//初始化 tpm 引脚配置和时钟
tpm_timer_init(TPM_0,3000,10);              //初始化 TPM0 模块 10ms 定时溢出
tpm_timer_init(TPM_1,3000,20);              //初始化 TPM1 模块 20ms 定时溢出
tpm_timer_init(TPM_2,3000,20);              //初始化 TPM2 模块 20ms 定时溢出
//初始化输入捕捉,采用 2 模块,0 通道
incap_init(TPM2_CH0,CAP_UP);                //通道连接在 PTA1
//初始化输出比较
outcompare_init(TPM0_CH0,5,CMP_REV);        //通道连接在 PTC1
//初始化 PWM 信号输出
pwm_init(TPM1_CH0,0.0,1,PWM_PLUS);          //通道连接在 PTA4
```

在"初始化外设模块"位置调用 TPM 构件中的使能模块中断函数：

```
tpm_enable_int(0);              //使能 TPM0 中断
tpm_enable_int(1);              //使能 TPM1 中断
tpm_enable_int(2);              //使能 TPM2 中断
```

在"开总中断"位置调用 common.h 文件中的开总中断宏函数：

```
ENABLE_INTERRUPTS;        //开总中断
```

这样,TPM 的输入捕捉输出比较测试工程初始化完成。

3）在 startup_MKL25Z4.S 文件的中断向量表中找到相应中断服务例程的函数名

在工程框架的"..\03_MCU\startup_MKL25Z4.S"文件中的中断向量表位置找到 TPM0、TPM1、TPM2 中断处理函数的默认函数名,分别是 TPM0_IRQHandler、TPM1_IRQHandler、TPM2_IRQHandler。

4）在 isr.c 文件中进行中断处理程序的编程

在工程框架的"..\08_Source\isr.c"文件中添加相应的中断处理函数并编程。

```
//==========================================================
//函数名称：TPM0_IRQHandler(TPM0 中断服务例程)
//功能概要：10ms 中断一次,本程序执行一次,静态变量 cnt 加 1,到达 100,即 1s 时间,
//          调用 SecAdd1,给全局变量数组 g_time 赋值(时、分、秒)
//==========================================================
void TPM0_IRQHandler(void)
{
    static uint_32 cnt;                       //中断次数
    if(tpm_get_int(0) == 1)                   //若有 TPM0 的溢出中断
    {
        tpm_clear_int(0);                     //清 TPM0 的溢出中断标志
        cnt++;                                //中断次数+1
        if(cnt >= 100)                        //若达到 100 次(即 1s)
        {
            cnt = 0;                          //中断次数清 0
            //调用"秒+1 计时子函数",给全局变量数组 g_time 赋值(时、分、秒)
```

```
                SecAdd1(g_time);
            }
        }
    }

    //==========================================================================
    //函数名称：TPM1_IRQHandler(TPM1中断服务例程)
    //功能概要：20ms中断一次,本程序执行一次,静态变量duty由0.0增到100.0,
    //          再由100.0减到0.0,然后根据duty值来改变pwm的占空比的值,
    //          对应的PWM输出引脚可以测得相应的PWM波形
    //==========================================================================
    void TPM1_IRQHandler(void)
    {
        static float duty=0.0;                //静态变量duty(占空比)
        static uint_8 Up_Down=1;              //占空比增减标志

        if(tpm_get_int(1) == 1)               //若有TPM1的溢出中断
        {
            tpm_clear_int(1);                 //清TPM1的溢出中断标志
            pwm_update(TPM1_CH0,duty);        //PWN更新
            if(Up_Down==1)                    //占空比逐渐增加
            {
                duty=duty+0.5;
                if(duty>100.0)                //防止占空比越界
                {
                    duty=100.0;
                    Up_Down=0;
                }
            }
            else                              //占空比逐渐减小
            {
                duty=duty-0.5;
                if(duty<0)                    //防止占空比越界
                {
                    duty=0;
                    Up_Down = 1;
                }
            }
        }
    }
    //==========================================================================
    //函数名称：TPM2_IRQHandler(TPM2中断服务例程)
    //功能概要：进入中断后判断是20ms定时器溢出中断,还是通道捕获中断,如果是计数器溢出
    //          中断清中断标志位后返回,如果是通道捕获中断那么记录当前捕获值,当记录
    //          满10个值后,将这10个值一并从串口打出
    //==========================================================================
    void TPM2_IRQHandler(void)
    {
        static uint_32 Cap_Value[10];         //捕获的通道值
        static uint_32 cnt=0;                 //捕捉次数
```

```
        //若有 TPM2 的溢出中断,清其中断标志
        if(tpm_get_int(2)==1)   tpm_clear_int(2);
        //有通道中断产生
        if(tpm_chl_get_int(2,0) == 1)                    //判断中断类型是否为通道中断
        {
            tpm_clear_chl_int(2,0);                      //清除通道中断标志
            Cap_Value[cnt]=tpm_get_capvalue(TPM2_CH0);   //获取通道捕获值
            cnt++;                                       //扑捉次数+1
            if(cnt>=10)                                  //连续获取 10 次捕获值
            {
                while(cnt)                               //将 10 个捕获值统一通过串口打出
                {
                    cnt--;
                    printf("TPM2 通道 0 的输入捕捉通道值: %d\n",Cap_Value[cnt]);
                }
            }
        }
    }
```

7.4　TPM 模块驱动构件的设计方法

7.4.1　TPM 模块的编程结构

每个 TPM 模块均有状态和控制寄存器(SC)、计数器(CNT)、模数寄存器(MOD)。另外,还有通道状态和控制寄存器(CnSC)、通道值寄存器(CnV)、捕捉和比较状态寄存器(STATUS)及配置寄存器(TPMx_CONF)等。有关地址在头文件中给出,这里不再列出。

1. TPM 模块的状态和控制寄存器

如表 7-4 所示,状态和控制寄存器(SC)包含的溢出状态标志和控制位,用于配置中断使能、模块配置和预分频因子。在这个模块内,这些控制位与所有的通道有关。该寄存器复位后为 0。

表 7-4　SC 结构

数据位	D15～D9	D8	D7	D6	D5	D4、D3	D2～D0
读	0	DMA	TOF	TOIE	CPWMS	CMOD	PS
写	—		W1C				

D31～D9——保留,读取为 0。

D8(DMA)——DMA 使能位。DMA=0,禁用 DMA 传输,DMA=1,使能 DMA 传输。

D7(TOF)——定时器溢出标志。当计数器 CNT 的值等于模数寄存器 MOD 的值,TOF=1。写 1 清 0 该位,写 0 无效。若写 1 清 0 期间,又发生定时器溢出,则 TOF 仍为 1。

D6(TOIE)——定时器溢出中断使能位,TOIE=0,禁用 TOF 中断,TOIE=1,使能 TOF 中断,即设定当 TOF 等于 1 时产生定时器溢出中断。

D5(CPWMS)——PWM中央对齐模式选择,这个模式配置TPM定时器计数模式为可逆计数模式。此字段是写保护的。仅当计数器禁用时它才可以被写入。CPWMS=0,计数器在上升计数模式运行。CPWMS=1,计数器在可逆计数模式运行。

D4、D3(CMOD)——时钟模式选择。当禁用计数器时,这个位段保持置位。CMOD=00,计数器禁用;CMOD=01,计数器在每一个计数周期增1;CMOD=10,计数器在TPM外部时钟的上升沿同步到计数器时钟时增1;CMOD=11,保留。

D2～D0(PS)——预分频因子。选择通过CMOD为时钟模式选择8个分频系数中的一个。此字段写保护。只有当计数器是禁用时才可以被写。记预分频因子为n,分频系数为m,则$m=2^n$。

2. 计数器

计数器(CNT)为16位计数器,复位值为0,向它写入任何值也可以使之清0。有两种计数模式:上升计数和可逆计数。上升计数用在一般的情况下,如:输入捕捉、边沿对齐的PWM信号输出等,可逆计数则主要用在中央对齐的PWM信号输出模式中。

上升计数: 当CPWMS=0时,上升计数被选中。0值被加载到TPM计数器中,并且计数器增量直到达到MOD中的值,此刻计数器被重载为0。当使用上升计数时,TPM周期是(MOD + 0x0001)×TPM的计数器时钟的周期。当TPM计数器从0变到MOD时,TOF位被置位。

可逆计数: 当CPWMS=1时,可逆计数被选中。当配置为可逆计数时,MOD必须大于等于2。0值被加载到TPM计数器,并且计数器增量直到达到MOD值,此时计数器减量直到它返回0值并且可逆计数重启。当使用可逆计数时,TPM周期是2×MOD×TPM计数器时钟周期。当TPM计数器从MOD变化到MOD−1时,TOF位被置位。

以所选48MHz时钟为例,计数器的最小计数值约为1/48MHz≈20.8ns。

3. 模数寄存器

模数寄存器(MOD)存放计数器的模值。含义是当计数器达到模值并且增加时产生溢出,TOF=1,并且下一个计数器的值取决于选定的计数方式,如果是单向计数则下一个值从初始值重新开始计数,如果是可逆计数则下一个值从当前值开始减少。写入MOD寄存器锁存值到缓冲区中,根据MOD寄存器的更新而更新写缓冲区的值。建议在初始化计数器(写CNT)之前写MOD寄存器来避免在第一个计数器溢出时发生混淆。

D31～D16——保留,读取为0。

D15～D0(MOD)——模值,当写这个字段时,该字段的两个字节必须被同时写入。

4. 通道(n)值寄存器

这个寄存器用于PWM、输入捕捉、输出比较。在输入捕捉情况下,该寄存器包含捕获计数器的值,CnV的写操作被忽略。在输出比较模式,写CnV寄存器将锁存值传到缓冲区。

D31～D16——保留,读取为0。

D15～D0(VAL)——通道值,捕捉计数器输入模式的值或输出模式的匹配值。在写这个字段时,该字段的两个字节必须被同时写入。

5. 通道(n)状态和控制寄存器

如表7-5所示,通道状态控制寄存器(CnSC)包含通道状态标志位以及用来配置中断使

能、通道模式的控制位，该寄存器复位值为 0。

表 7-5　CnSC 结构

数据位	D31～D8	D7	D6	D5	D4	D3	D2	D1	D0
读	0	CHF	CHIE	MSB	MSA	ELSB	ELSA	0	DMA
写	—	W1C						—	

D31～D8，D1——保留，读取为 0。

D7(CHF)——通道标志，当通道上有事件发生时，CHF＝1，这一位通过写 1 来清 0。如果另一个事件发生在 CHF 置位和写操作之间时，写操作无效；因此，CHF 保留置位表明另一个事件已经发生。由于清 0 前一个 CHF 位时存在延迟，因此在这种情况下 CHF 中断请求不会丢失。CHF＝0，没有通道事件发生，CHF＝1，有通道事件发生。

D6(CHIE)——通道中断使能。CHIE＝0，禁止通道中断，CHIE＝1，使能通道中断。

D5～D2(MSB、MSA、ELSB、ELSA)——与状态和控制寄存器 SC 的 PWM 中央对齐模式选择位 CPWMS 配合完成通道模式设置，见表 7-6。

表 7-6　模式、边沿和电平选择

CPWMS	MSnB：MSnA	ELSnB：ELSnA	模　式	配　置
X	00	00	无	通道禁止
X	01/10/11	00	软件比较	TPM 不使用引脚
0	00	01	输入捕捉	上升沿捕捉
		10		下降沿捕捉
		11		在上升沿或者下降沿捕捉
	01	01	输出比较	翻转比较输出
		10		输出低电平
		11		输出高电平
	10	10	边沿对齐 PWM	前半段高电平，后半段低电平
		X1		前半段低电平，后半段高电平
	11	10	输出比较	输出低电平
		X1		输出高电平
1	10	10	中心对齐 PWM	向上计数输出低电平，向下计数输出高电平
		X1		向上计数输出高电平，向下计数输出低电平

D0(DMA)——使能通道 DMA 传输。DMA＝0，禁用 DMA，DMA＝1，使能 DMA。

边沿对齐 PWM(Edge-Aligned PWM)编程方法介绍：如图 7-5 所示，当 CPWMS＝0，MSnB：MSnA＝1：0 时，边缘对齐方式被选中。EPWM 周期由 MOD ＋ 0x0001 决定，并且脉冲宽度(占空比)由 CnV 决定。在通道(n)匹配 TPM 计数器＝CnV 时，CHnF 位被置位并且通道(n)中断产生(如果 CHnIE＝1)，也就是在脉冲宽度的末尾。因为所有 PWM 信号的主要边沿都对齐到周期的开始，所以这种类型的 PWM 信号被称为边沿对齐，在一个 TPM 中所有的通道也是如此。

PWM 中央对齐输出配置方式如图 7-6 所示。

中央对齐 PWM(Center-Aligned PWM)配置方式介绍：如图 7-6 所示，CPWM 周期由

计数器溢出　计数器溢出　计数器溢出
周期
脉冲宽度
通道(n)输出
通道(n)匹配　通道(n)匹配　通道(n)匹配

图 7-5　边沿对齐 PWM

2×MOD 决定，脉冲宽度(占空比)由 2×CnV 决定，MOD 必须在 0x0001～0x7FFF 的范围内。

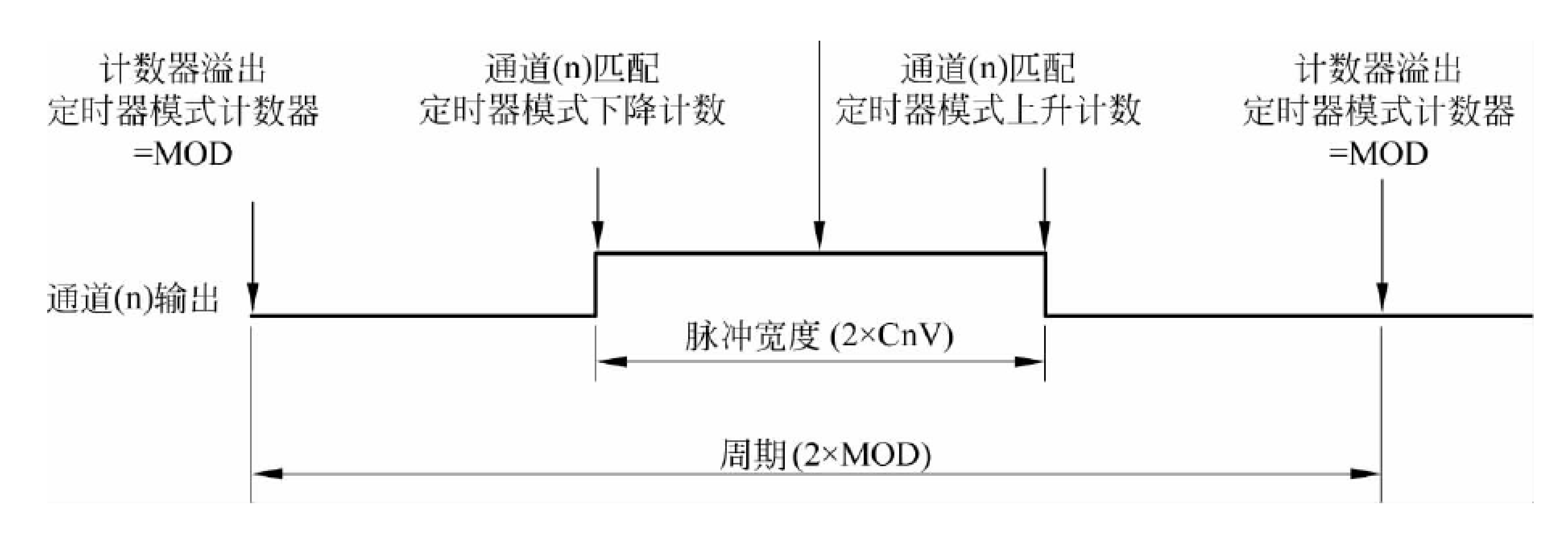


图 7-6　中央对齐 PWM

6. 输入捕捉和输出比较状态寄存器

对于每个 TPM 通道中，输入捕捉和输出比较状态寄存器包含状态标志 CHnF 位(在 CnSC 中)和 TOF 位(在 SC 中)的一个备份，这是为了便于软件编写。在状态寄存器中的每个 CHnF 位是 CnSC 中 CHnF 位的一个镜像。所有 CHnF 位可以被检查仅需一次读状态操作。所有 CHnF 位可以通过向状态寄存器写所有的位来清 0。当通道事件发生时，硬件设置独立的通道标志。写 1 到 CHF 位可以清 0，写 0 无效。如果另一个事件发生在标志置位和写操作之间时，写操作无效。因此，CHF 保持置位表明另一个事件已经发生。在这种情况下，由于前一个 CHF 的清零序列存在，所以 CHF 中断请求不会丢失。该寄存器复位值为 0，其结构如表 7-7 所示。

表 7-7　STATUS 结构

数据位	D31～D9	D8	D7、D6	D5	D4	D3	D2	D1	D0
读	0	TOF	0	CH5F	CH4F	CH3F	CH2F	CH1F	CH0F
写	—	W1C		W1C	W1C	W1C	W1C	W1C	W1C

D31～D9，D7，D6——保留，读取为 0。

D8(TOF)——定时器溢出标志，TOF=0，计数器未溢出。TOF=1，计数器溢出。

D5(CH5F)～D0(CH0F)——通道 5～通道 0 标志，CHxF=0，x 通道没有事件产生。CHxF=1，x 通道事件已经产生。

7. 配置寄存器

该寄存器复位值为 0,结构如表 7-8 所示。

表 7-8　CONF 结构

数据位	D31～D28	D27～D24	D23～D19	D18	D17	D16	D15～D10	D9	D8	D7、D6	D5	D4～D0
读	0	TRGSEL	0	CROT	CSOO	CSOT	0	GTBEEN	0	DBGMODE	DOZEEN	0
写	—		—				—		—			—

D31～D28,D23～D19,D15～D10,D8,D4～D0——保留,读取为 0。

D27～D24(TRGSEL)——触发选择。选择输入触发用于启动/重载计数器。这个位段只有当计数器是禁用时才可以被改变。

D18(CROT)——计数器重载触发,当置位并且在选择的触发输入上检测到上升沿时,计数器将重载为零(并且初始化 PWM 输出为其默认值)。如果计数器在调试模式或睡眠模式下被暂停时,触发输入被忽略。这个位段只有当计数器是禁用时才能被改变。CROT=0,当在选择的输入触发上检测到上升沿不会进行计数器重载;CROT=1,当在选择的输入触发上检测到上升沿时,计数器重载。

D17(CSOO)——计数器在溢出时停止,当置位时,一旦计数器值等于 MOD 值并且递增(这也置位 TOF),那么计数器将会停止增量。由于写计数器寄存器或者一个触发输入不会导致计数器停止增量,所以计数器重载为 0。一旦计数器停止增量,那么除非它先禁止在使能或者当 CSOT 置位时,在已选定的触发输入上检测到上升沿,否则计数器不会开始递增。这个位段应该只有当计数器是禁用时才可以被改变。CSOO=0,在溢出之后,计数器继续增量/减量。CSOO=1,在溢出之后,计数器停止增量/减量。

D16(CSOT)——计数器在触发时开始,当置位时,计数器在使能后将不会增量直到在已选择的触发输入上检测到上升沿。如果计数器由于溢出而停止,那么在已选择的触发输入的上升沿将导致计数器再次开始增量。如果计数器在调试模式或者睡眠模式被暂停的话,那么触发输入将被忽略。这一位段应该只有在计数器是禁用时才可以被改变。CSOT=0,一旦被使能,计数器立即开始增量。CSOT=1,当被使能或者由于溢出被停止之后,当在已选择的触发输入上检测到上升沿时,计数器开始增量。

D9(GTBEEN)——全局时基使能,配置 TPM 使用一个外部产生全局时基计数器。当一个外部产生的时基被使用,内部计数器不会被用于通道,但可以用来生成一个周期中断或 DMA 请求使用模寄存器和定时器溢出标志。GTBEEN=0,所有通道使用内部产生的计数器作为它们的时基。GTBEEN=1,所有通道使用一个外部产生的全局时基作为它们的时基。

D7、D6(DBGMODE)——调试模式,在调试模式下配置 TPM 的行为。所有其他配置被保留。DBGMODE=00,在调试模式下计数器被暂停并且不会增量。触发输入和输入捕捉事件也被忽略了。DBGMODE=11,调试模式下计数器继续。

D5(DOZEEN)——睡眠使能,在等待模式配置 TPM 行为。DOZEEN=0,在睡眠模式下内部计数器继续。DOZEEN=1,在睡眠模式下内部计数器被暂停并且不会增量。触发输入和输入捕捉事件也被忽略。

7.4.2 TPM驱动构件的设计

TPM驱动构件存放于工程目录“..\02-Software\KL25共用驱动\KL25底层驱动构件”文件夹中，供复制使用，各个工程文件夹下的“..\05_Driver\tpm”文件夹中TPM驱动构件与此一致。TPM驱动构件的实现在源程序文件tpm.c中。下面给出源程序文件tpm.c中与寄存器相关的主要函数内容。

TPM的时钟由SIM_SOPT2[TPMSRC]和SIM_SOPT2[PLLFLLSEL]来进行选择，TPMSRC[25:24]=00表示没有选择任何时钟，相当于关闭TPM计数器；TPMSRC[25:24]=01表示选择MCGFLLCLK时钟或者MCGPLLCLK/2作为时钟；TPMSRC[25:24]=10表示选择OSCERCLK时钟；TPMSRC[25:24]=11表示选择MCGIRCL作为时钟。

选择的时钟源的分频因子由状态和控制(TPMx_SC)的PS[2:0]位决定，PS[2:0]=000(1分频)，PS[2:0]=001(2分频)，PS[2:0]=010(4分频)，PS[2:0]=011(8分频)，PS[2:0]=100(16分频)，PS[2:0]=101(32分频)，PS[2:0]=110(64分频)，PS[2:0]=111(128分频)。TPM构件源文件(tpm.c)如下。

```
//============================================================
//文件名称：tpm.c
//功能概要：tpm底层驱动构件源文件
//============================================================
#include "tpm.h"

//各端口基地址放入常数数据组PORT_ARR[0]~PORT_ARR[4]中
static const PORT_MemMapPtr PORT_ARR[]={PORTA_BASE_PTR,PORTB_BASE_PTR,
                                        PORTC_BASE_PTR,PORTD_BASE_PTR,PORTE_BASE_PTR};
//定时器模块0,1,2地址映射
TPM_MemMapPtr TPM_ARR[]={TPM0_BASE_PTR,TPM1_BASE_PTR,TPM2_BASE_PTR};
IRQn_Type TPM_IRQ[]={TPM0_IRQn,TPM1_IRQn,TPM2_IRQn};

static uint_8 tpm_mux_val(uint_16 tpmx_Chy,uint_8 * TPM_i,uint_8 * chl);

//============================================================
//函数名称：tpm_timer_init
//功能概要：tpm模块初始化，设置计数器频率f及计数器溢出时间MOD_Value。
//参数说明：TPM_i：模块号，使用宏定义：TPM_0、TPM_1、TPM_2
//          f：单位：kHz，取值：375、750、1500、3000、6000、12000、24000、48000
//          MOD_Value：单位ms，范围取决于计数器频率与计数器位数(16位)
//函数返回：无
//============================================================
void tpm_timer_init(uint_16 TPM_i,uint32_t f,float MOD_Value)
{
    //局部变量声明
    uint_32 clk_f,clk_div;
    //1. 开启SIM时钟门
    BSET(SIM_SCGC6_TPM0_SHIFT+TPM_i,SIM_SCGC6);
    //2.开启时钟，默认选择用PLL/2时钟源，即48MHz
    SIM_SOPT2 |= SIM_SOPT2_TPMSRC(1);
```

```
    SIM_SOPT2 |= SIM_SOPT2_PLLFLLSEL_MASK;
    //3. 由期望的时钟频率 f,计算分频因子 clk_div。因分频系数 clk_f=48000/f,
    //则分频因子 clk_div=sqrt(clk_f)。例如: f=3000kHz,则 clk_f=16,clk_div=4
    clk_f=48000/f;
    while(clk_f>1)
    {
        clk_div++;
        clk_f=clk_f/2;
    }
    //4. 写 TPM_i 的状态和控制寄存器
    TPM_ARR[TPM_i]->SC |= TPM_SC_TOIE_MASK|TPM_SC_CMOD(1)| \
                          TPM_SC_PS(clk_div);
    //5. 计数器清零
    TPM_ARR[TPM_i]->CNT = 0;
    //6. 设置模数寄存器
    TPM_ARR[TPM_i]->MOD = f * MOD_Value;//给模数寄存器赋值
}
//===========================================================================
//函数名称: pwm_init
//功能概要: pwm 模块初始化
//参数说明: tpmx_Chy: 模块通道号(例: TPM_CH0 表示为 TPM0 模块第 0 通道)
//        duty: 占空比: 0.0~100.0 对应 0%~100%
//          Align: PWM 计数对齐方式(有宏定义常数可用)
//        pol: PWM 极性选择(有宏定义常数可用)
//函数返回: 无
//===========================================================================
void pwm_init(uint_16 tpmx_Chy,float duty,uint_8 Align,uint_8 pol)
{
    PORT_MemMapPtr port_ptr;            //声明 port_ptr 为 PORT 结构体类型指针
    uint_8 port,pin;                    //解析出的端口、引脚号临时变量
    uint_8 mux_val;
    uint_8 TPM_i,chl;           //由 tpmx_Chy 解析出的 tpm 模块号、通道号临时变量
    uint_32 period;
    // 防止越界
    if(duty>100.0) duty=100.0;
    //1. gpio 引脚解析
    port = (tpmx_Chy>>8);               //解析出的端口
    pin = tpmx_Chy;                     //解析出的引脚号
    //2. 根据 port,给局部变量 port_ptr 赋值,获得基地址
    port_ptr = PORT_ARR[port];          //获得 PORT 模块相应口基地址
    //3. 取得引脚复用值,并获得解析的 tpm 模块号和通道号
    mux_val=tpm_mux_val(tpmx_Chy,&TPM_i,&chl);
    //4. 根据 pin,指定该引脚功能为 TPM 的通道功能(即令引脚控制寄存器的 MUX=mux_val)
    PORT_PCR_REG(port_ptr, pin) |= PORT_PCR_MUX(mux_val);
    //5. PWM 对齐方式的设定
    if(Align == PWM_CENTER)             //中心对齐
        TPM_ARR[TPM_i]->SC |= TPM_SC_CPWMS_MASK;
    else                                //边沿对齐
        TPM_ARR[TPM_i]->SC &= ~TPM_SC_CPWMS_MASK;
```

```
    //6. PWM极性的设定
    if(pol == PWM_PLUS)                        //正极性
        TPM_CnSC_REG(TPM_ARR[TPM_i],chl)=TPM_CnSC_MSB_MASK| \
                                        TPM_CnSC_ELSB_MASK;
    else                                       //负极性
        TPM_CnSC_REG(TPM_ARR[TPM_i],chl)=TPM_CnSC_MSB_MASK| \
                                        TPM_CnSC_ELSA_MASK;
    //7. PWM占空比的设定
    period=TPM_ARR[TPM_i]->MOD;        //计算周期(period)
    TPM_CnV_REG(TPM_ARR[TPM_i],chl)=(uint_32)(period * duty/100);
}

//==========================================================
//函数名称: pwm_update
//功能概要: tpmx模块Chy通道的PWM更新
//参数说明: tpmx_Chy: 模块通道号(例: TPM_CH0表示为TPM0模块0通道)
//          duty: 占空比: 0.0~100.0对应0%~100%
//函数返回: 无
//==========================================================
void pwm_update(uint_16 tpmx_Chy,float duty)
{
    uint_8 TPM_i,chl;                        //由tpmx_Chy解析出的tpm模块号、通道号临时变量
    uint_32 period;
    // 防止越界
    if(duty>100.0)   duty=100.0;
    //1. 取得引脚复用值,并获得解析的tpm模块号和通道号
    tpm_mux_val(tpmx_Chy, &TPM_i, &chl);
    //2. 更新PWM通道寄存器值
    period=TPM_ARR[TPM_i]->MOD;
    TPM_CnV_REG(TPM_ARR[TPM_i],chl)=(uint32_t)period * duty/100;
}

//==========================================================
//函数名称: incap_init
//功能概要: incap模块初始化
//参数说明: tpmx_Chy: 模块通道号(例: TPM_CH0表示为TPM0模块第0通道)
//          capmode: 输入捕捉模式(上升沿、下降沿、双边沿),有宏定义常数使用
//函数返回: 无
//==========================================================
void incap_init(uint_16 tpmx_Chy,uint_8 capmode)
{
    PORT_MemMapPtr port_ptr;              //声明port_ptr为PORT结构体类型指针
    uint_8 port,pin;                      //解析出的端口、引脚号临时变量
    uint_8 TPM_i,chl;                     //由tpmx_Chy解析出的tpm模块号、通道号临时变量
    uint_8 mux_val;

    //1. gpio引脚解析
    port = (tpmx_Chy>>8);                 //解析出的端口
    pin = tpmx_Chy;                       //解析出的引脚号
```

```
    //2．根据 port,给局部变量 port_ptr 赋值,获得基地址
    port_ptr = PORT_ARR[port];          //获得 PORT 模块相应口基地址
    //3．取得引脚复用值,并获得解析的 tpm 模块号和通道号
    mux_val=tpm_mux_val(tpmx_Chy,&TPM_i,&chl);
    //4．根据 pin,指定该引脚功能为 TPM 的通道功能(即令引脚控制寄存器的 MUX=mux_val)
    PORT_PCR_REG(port_ptr, pin) |= PORT_PCR_MUX(mux_val);
    //5．输入捕捉参数设定
    if(capmode == CAP_UP)               //上升沿捕捉
        TPM_CnSC_REG(TPM_ARR[TPM_i],chl) = TPM_CnSC_ELSA_MASK;
    else if(capmode == CAP_DOWN)        //下降沿捕捉
        TPM_CnSC_REG(TPM_ARR[TPM_i],chl) = TPM_CnSC_ELSB_MASK;
    else if(capmode == CAP_DOUBLE)      //双边沿捕捉
    {
        TPM_CnSC_REG(TPM_ARR[TPM_i],chl) = TPM_CnSC_ELSA_MASK;
        TPM_CnSC_REG(TPM_ARR[TPM_i],chl) = TPM_CnSC_ELSB_MASK;
    }
    //6．chl 通道中断使能
    TPM_CnSC_REG(TPM_ARR[TPM_i],chl) |= TPM_CnSC_CHIE_MASK;
}

//======================================================================
//函数名称：tpm_get_capvalue
//功能概要：获取 tpmx 模块 Chy 通道的计数器当前值
//参数说明：tpmx_Chy：模块通道号(例：TPM_CH0 表示为 TPM0 模块第 0 通道)
//函数返回：tpmx 模块 Chy 通道的计数器当前值
//======================================================================
uint_16 tpm_get_capvalue(uint_16 tpmx_Chy)
{
    uint_8 TPM_i,chl;                   //由 tpmx_Chy 解析出的 tpm 模块号、通道号临时变量
    uint_16 cnt;
    tpm_mux_val(tpmx_Chy,&TPM_i,&chl);      //解析 tpm 模块号和通道号
    cnt=TPM_CnV_REG(TPM_ARR[TPM_i],chl);
    return cnt;

}

//======================================================================
//函数名称：outcompare_init
//功能概要：outcompare 模块初始化
//参数说明：tpmx_Chy：模块通道号(例：TPM_CH0 表示为 TPM0 模块第 0 通道)
//          comduty：输出比较电平翻转位置占总周期的比例：0.0～100.0 对应 0%～100%
//          cmpmode：输出比较模式(翻转电平、强制低电平、强制高电平),有宏定义常数使用
//函数返回：无
//======================================================================
void outcompare_init(uint_16 tpmx_Chy,float comduty,uint_8 cmpmode)
{
    PORT_MemMapPtr port_ptr;            //声明 port_ptr 为 PORT 结构体类型指针
    uint_8 port,pin;                    //解析出的端口、引脚号临时变量
    uint_8 mux_val;
    uint_8 TPM_i,chl;                   //由 tpmx_Chy 解析出的 tpm 模块号、通道号临时变量
```

```
    uint_32 mod;

    if(comduty>100.0)   comduty=100.0;

    //1. gpio引脚解析
    port = (tpmx_Chy>>8);                    //解析出的端口
    pin = tpmx_Chy;                          //解析出的引脚号
    //2. 根据port,给局部变量port_ptr赋值,获得基地址
    port_ptr = PORT_ARR[port];               //获得PORT模块相应口基地址
    //3. 取得引脚复用值,并获得解析的tpm模块号和通道号
    mux_val=tpm_mux_val(tpmx_Chy,&TPM_i,&chl);
    //4. 根据pin,指定该引脚功能为TPM的通道功能(即令引脚控制寄存器的MUX=mux_val)
    PORT_PCR_REG(port_ptr, pin) |= PORT_PCR_MUX(mux_val);
    //5. 输出比较模式的设定
    if(cmpmode == CMP_REV)                   //翻转模式
      TPM_CnSC_REG(TPM_ARR[TPM_i],chl)=TPM_CnSC_MSA_MASK| \
                                       TPM_CnSC_ELSA_MASK;
    else if(cmpmode == CMP_LOW)              //强制低电平模式
      TPM_CnSC_REG(TPM_ARR[TPM_i],chl)=TPM_CnSC_MSA_MASK| \
              TPM_CnSC_ELSB_MASK;
    else if(cmpmode == CMP_HIGH)             //强制高电平模式
      TPM_CnSC_REG(TPM_ARR[TPM_i],chl)=TPM_CnSC_MSA_MASK| \
                   TPM_CnSC_ELSA_MASK|TPM_CnSC_ELSB_MASK;
    //6. 输出比较占空比的设定
    mod=TPM_ARR[TPM_i]->MOD;                 //读取模数寄存器的值(MOD)
    TPM_CnV_REG(TPM_ARR[TPM_i],chl)=(uint32_t)mod * (comduty/100);
}

//=========================================================================
//函数名称: tpm_enable_int
//功能概要: 使能tpm模块中断
//参数说明: TPM_i: 模块号,使用宏定义: TPM_0、TPM_1、TPM_2
//函数返回: 无
//=========================================================================
void tpm_enable_int(uint_8 TPM_i)
{
    if(TPM_i>2)   TPM_i = 2;
    //开TPM中断
    NVIC_EnableIRQ(TPM_IRQ[TPM_i]);
}

//=========================================================================
//函数名称: tpm_disable_int
//功能概要: 禁用tpm模块中断
//参数说明: TPM_i: 模块号,使用宏定义: TPM_0、TPM_1、TPM_2
//函数返回: 无
//=========================================================================
void tpm_disable_int(uint_8 TPM_i)
{
    if(TPM_i>2)   TPM_i = 2;
```

```
    //开 TPM 中断
    NVIC_DisableIRQ(TPM_IRQ[TPM_i]);
}

//==========================================================
//函数名称：tpm_get_int
//功能概要：获取 TPM 模块中断标志
//参数说明：TPM_i：模块号,使用宏定义：TPM_0、TPM_1、TPM_2
//函数返回：中断标志 1=有中断产生;0=无中断产生
//==========================================================
uint_8 tpm_get_int(uint_8 TPM_i)
{
  //获取 TPM_i 模块中断标志位
  if(((TPM_SC_REG(TPM_ARR[TPM_i]) & TPM_SC_TOF_MASK)==TPM_SC_TOF_
MASK))
    return 1;
  else
    return 0;
}

//==========================================================
//函数名称：tpm_chl_get_int
//功能概要：获取 TPM 通道中断标志
//参数说明：TPM_i：模块号,使用宏定义：TPM_0、TPM_1、TPM_2,TPMC_i：0～5 通道
//函数返回：中断标志 1=有中断产生;0=无中断产生
//==========================================================
uint_8 tpm_chl_get_int(uint_8 TPM_i,uint_8 TPMC_i)
{
   //获取 TPM_i 模块 TPMC_i 通道中断标志位
   if(((TPM_CnSC_REG(TPM_ARR[TPM_i],TPMC_i)&TPM_CnSC_CHF_MASK)\
                   ==TPM_CnSC_CHF_MASK))
       return 1;
   else
       return 0;
}

//==========================================================
//函数名称：tpm_clear_int
//功能概要：清除 TPM 中断标志
//参数说明：TPM_i：模块号,使用宏定义：TPM_0、TPM_1、TPM_2
//函数返回：清除中断标志位
//==========================================================
void tpm_clear_int(uint_8 TPM_i)
{
    //清除 TPM_i 模块中断标志位
    BSET(TPM_SC_TOF_SHIFT,TPM_SC_REG(TPM_ARR[TPM_i]));
}
```

```
//=====================================================================
//函数名称：tpm_clear_chl_int
//功能概要：清除 TPM 通道中断标志
//参数说明：TPM_i: 模块号,使用宏定义：TPM_0、TPM_1、TPM_2,TPMC_i: 0～5 通道
//函数返回：清除 TPM 通道中断标志位
//=====================================================================
void tpm_clear_chl_int(uint_8 TPM_i,uint_8 TPMC_i)
{
    //清除 TPM_i 模块 TPMC_i 通道中断标志位
    BSET(TPM_CnSC_CHF_SHIFT,TPM_CnSC_REG(TPM_ARR[TPM_i],TPMC_i));
}

//--------以下为内部函数--------
//=====================================================================
//函数名称：tpm_mux_val
//功能概要：将传进参数 tpmx_Chy 进行解析,得出具体模块号与通道号(例：TPM_CH0
//          解析出 PORT 引脚,并根据引脚返回 mux_val)
//参数说明：tpmx_Chy：模块通道号(例：TPM_CH0 表示为 TPM0 模块第 0 通道)
//
//函数返回：gpio 复用寄存器传入值
//=====================================================================
static uint_8 tpm_mux_val(uint_16 tpmx_Chy,uint_8 * TPM_i,uint_8 * chl)
{
    uint_8 port,pin;
    //1.解析模块号和通道号
    switch(tpmx_Chy)
    {
        case TPM0_CH0: * TPM_i =0; * chl=0;break;
        case TPM0_CH1: * TPM_i =0; * chl=1;break;
        case TPM0_CH2: * TPM_i =0; * chl=2;break;
        case TPM0_CH3: * TPM_i =0; * chl=3;break;
        case TPM0_CH4: * TPM_i =0; * chl=4;break;
        case TPM0_CH5: * TPM_i =0; * chl=5;break;
        case TPM1_CH0: * TPM_i =1; * chl=0;break;
        case TPM1_CH1: * TPM_i =1; * chl=1;break;
        case TPM2_CH0: * TPM_i =2; * chl=0;break;
        case TPM2_CH1: * TPM_i =2; * chl=1;break;
    }
    //2.解析引脚复用寄存器传入值
    port = (tpmx_Chy>>8);
    pin = tpmx_Chy;
    if(port<2 || port==4 || (port==2 && (pin==8||pin==9) ))
        return 3;
    else
        return 4;
}
```

7.5　周期中断定时器 PIT 模块

7.5.1　周期中断定时器 PIT 模块功能概述

KL25/26 内部有一个周期中断定时器模块(Periodic Interrupt Timer,PIT)模块,内含两个通道,没有外部引脚。每个通道都有一个独立的 32 位的减 1 计数的计数器(CVALn,n=0,1),时钟源固定为系统总线时钟并且不可分频。当 PIT 模块的某一通道被初始化使能后,计数器 CVALn 的值会自动加载重载寄存器 LDVALn 中的值,开始按照时钟源频率减 1 计数,到 0 时,标志寄存器 TFLGn 的 TIF 位被置 1,产生定时溢出中断。对应的中断向量号为 38,非内核中断请求 IRQ 号为 22。关于中断周期的最大值可由时钟源频率及计数器位数计算获得。例如,时钟源频率为 24MHz,可计算得出中断周期最大值为 178s,实际的中断周期由重载寄存器 LDVALn 值决定。

7.5.2　PIT 驱动构件及使用方法

1. PIT 驱动构件基本要素分析

PIT 模块有两个独立的通道,时钟源为系统总线时钟,为方便使用,在头文件中给出通道号及工作时钟频率的宏定义。一般来说,在中断服务例程中,需要判断某一通道计数器是否产生溢出中断,因此,在头文件中给出了宏函数 PIT_GET_FLAG(index),用于判断 index 通道计数器是否产生溢出中断,其中,参数 index 为通道号。头文件中还给出了宏函数 PIT_CLEAR_FLAG(index),用于清中断标志。除宏定义及宏函数之外,头文件中还需给出对外接口函数:PIT 模块初始化函数 pit_init,形参为通道号及以毫秒为单位的中断周期;使能中断函数 pit_enable_int 及禁止中断函数 pit_disable_int,它们的形参为通道号。这样可以满足 PIT 模块的基本编程。

2. PIT 驱动构件头文件

```
//======================================================================
//文件名称:pit.h
//功能概要:KL25 pit 底层驱动程序头文件
//版权所有:苏州大学恩智浦嵌入式中心(sumcu.suda.edu.cn)
//更新记录:2016-3-20    V4.0
//======================================================================
#ifndef _PIT_H
#define _PIT_H
//1 头文件
#include "common.h"

//2 宏定义
#define CH_0                0
#define CH_1                1
#define PIT_WORK_FREQ       BUS_CLK_KHZ
//获取中断标志
```

```
#define PIT_GET_FLAG(index)        (PIT_TFLG(index) & PIT_TFLG_TIF_MASK) == \
                                   PIT_TFLG_TIF_MASK)
//清中断标志
#define PIT_CLEAR_FLAG(index)      (PIT_TFLG(index) |= PIT_TFLG_TIF_MASK)

//==========================================================================
//函数名称: pit_init
//函数返回: 无
//参数说明: channel:PIT 模块的通道号,0 或 1
//          freq:系统总线时钟频率,单位 kHz。例:系统总线时钟为 24MHz,则 freq=24 000
//          int_ms:中断周期,以 ms 为单位。系统总线时钟为 24MHz,最大值为 178 956ms
//功能概要: PIT 模块初始化
//调用举例: pit_init(CH_0,PIT_WORK_FREQ,1000);即初始化 PIT 模块的 0 通道使用总线
//时钟频率,中断周期为 1s
//==========================================================================
void pit_init(uint_8 channel,uint_16 freq,uint_32 int_ms);

//==========================================================================
//函数名称: pit_enable_int
//参数说明: channel:PIT 模块的通道号,0 或 1
//函数返回: 无
//功能概要: 使能某一通道的 PIT 中断
//调用举例: pit_enable_int(CH_0);使能 PIT 模块的通道 0 中断
//==========================================================================
void pit_enable_int(uint_8 channel);

//==========================================================================
//函数名称: pit_disable_int
//参数说明: channel:PIT 模块的通道号,0 或 1
//函数返回: 无
//功能概要: 禁止某一通道的 PIT 中断
//调用举例: pit_disable_int(CH_0);禁止 PIT 模块的通道 0 中断
//==========================================================================
void pit_disable_int(uint_8 channel);

#endif
```

3. PIT 驱动构件使用方法

设使用 PIT 模块的通道 0 进行计时,时钟源为系统总线时钟,中断周期为 1000ms,使用步骤如下。

(1) 在 main()函数的初始化外设模块位置添加下列语句:

```
pit_init(CH_0, PIT_WORK_FREQ, 1000);      //初始化 CH_0 使用系统总线时钟频率,周期为 1s
```

(2) 在 main()函数的使能模块中断位置添加下列语句:

```
pit_enable_int(CH_0);                     //使能 PIT 模块的 0 通道中断
```

(3) 在 isr.c 的中断服务例程 PIT_IRQHandler 中进行计时。g_time 为记录时分秒的全局变量数组,每中断一次,秒值加 1。函数 SecAdd1 的实现见工程目录下的源文件 common.c。

```
//==========================================================
//函数名称: PIT_IRQHandler
//函数返回: 无
//参数说明: 无
//功能概要: PIT 中断服务例程,清中断标志,并使用通道 0 完成计时,显示 MCU 运行时间
//调用举例: 无
//==========================================================
void PIT_IRQHandler(void)
{
    if(PIT_GET_FLAG(CH_0)
    {
        SecAdd1(g_time);                    //g_time 是时分秒全局变量数组
        PIT_CLEAR_FLAG(CH_0);               //清标志
    }
    else if((PIT_GET_FLAG(CH_1))
    {
        PIT_CLEAR_FLAG(CH_1);               //清标志
    }
}
```

4. PIT 驱动构件测试实例

PIT 构件的测试工程位于网上教学资源中的"..\program\CH07-KL25-PIT"文件夹。测试工程功能概述如下。

(1) 串口通信格式:波特率 9600,1 位停止位,无校验。

(2) PIT 使用系统总线时钟 24MHz。

(3) 上电或按复位按钮时,调试串口输出"苏州大学嵌入式实验室 PIT 构件测试用例!"。

(4) PIT 每 1s 中断一次,并在 PIT 中断中进行计时,调试串口每秒输出"MCU 记录的相对时间:00:00:01","00:00:01"为中断记录的时间,同时蓝色指示灯闪烁一次。

7.5.3　PIT 驱动构件设计

本节主要介绍如何根据 PIT 模块的各个寄存器的功能,结合上文给出的 pit.h 编写具体的 PIT 的驱动。

1. PIT 模块的编程结构

KL25 的 PIT 模块共有 11 个 32 位寄存器,包括一个模块控制寄存器(PIT_MCR),链接计数器高位寄存器(PIT_LTMR64H),链接计数器低位寄存器(PIT_LTMR64L)及通道 0、1 的寄存器组成。通道 0 包括一个重载寄存器(PIT_LDVAL0)、计数器(PIT_CVAL0)、控制寄存器(PIT_TCTRL0)、标志寄存器(PIT_TFLG0),通道 1 同样由 4 个相同功能的寄存器组成。通过对这些寄存器的编程,就可以使用 PIT 模块进行定时。有关 PIT 模块的存储器映像见 KL25 参考手册的 575 页。

1）模块控制寄存器(PIT_MCR)

该寄存器用于控制PIT定时器时钟及调试模式下的运行状态，仅高两位被使用，D29～D0位为保留位段，不可写，复位值为0。

D31(FRZ)——停止运行。当MCU进入调试模式时，允许定时器停止。FRZ=0，定时器在MCU调试模式下继续运行。FRZ=1，定时器在MCU调试模式下停止运行。复位值为0。

D30(MDIS)——模块禁用。禁用标准时钟，必须在模块其他设置完成之前通过该位完成模块时钟使能。MDIS=0，使能PIT定时器时钟；MDIS=1，禁用PIT定时器时钟。复位值为1。

2）控制寄存器(PIT_TCTRLn)

该寄存器包含每个定时器的所有控制位，仅高三位被使用，D28～D0位为保留位段，不可写，复位值为0。

D31(TEN)——定时器使能。使能或禁用定时器。TEN=0禁用定时器n；TEN=1使能定时器n。

D30(TIE)——定时器中断使能。有中断被挂起或者TFLGn寄存器的TIF位已经置位，使能该中断位会立即触发中断。为了避免这种情况发生，对应通道的TFLGn寄存器的TIF位必须先被清零。TIE=0，禁用定时器n的中断请求；TIE=1，一旦TIF置位，将产生中断请求。

D29(CHN)——链接模式。当被激活时，定时器n-1每次减至0之后，定时器n开始减1。定时器0无法工作在链接模式下。CHN=0，定时器未被链接；CHN=1，定时器被链接到前一个定时器。例如，对于通道2，如果这个位段置位，那么定时器2链接到定时器1。

3）标志寄存器(PIT_TFLGn)

32位的标志寄存器PIT_TFLGn(n可取0或1)只有D31有用，是定时器中断标志位(TIF)，该位为1，表示定时时间到，为0，表示定时时间未到。复位值为0，写1即可清0(w1c)。

4）重载寄存器(PIT_LDVALn)

D31～D0(TSV)——定时器定时值。设置定时器定时值，也是计数的最大初始值。定时器减1计数到0，然后它将触发一次中断并再次载入该值。写一个新值到这个寄存器将不会重启计数器，而是在当前计数值计时结束之后开始载入新值。要想终止当前计数并启动新值对应的计数周期，必须先禁用并重新使能计数器。

5）计数器(PIT_CVALn)

D31～D0(TVL)——定时器的当前计数值。使能定时器后，指明定时器的当前计数值。注意，如果禁用定时器，那么不要使用这个不可靠的值。定时器采用减1计数的方式工作。如果置位MCR寄存器的FRZ位，那么在调试模式下定时器的值不会改变，因为此时定时器已经停止运行。

6）链接计数器寄存器(PIT_LTMR64H和PIT_LTMR64L)

PIT模块还支持两个通道的定时器进行链接，即两个32位的定时器拼接成一个64位长的“链接定时器”，定时器n-1为低32位，定时器n为高32位。当PIT模块工作在链接

模式时，定时器 n−1 每次减至 0 之后，定时器 n 开始减 1。此处 n 不可为 0，因为 KL25 的定时器 0 无法工作在链接模式下，此处仅为阐述“链接定时器”的概念。

链接计数器寄存器由两个寄存器组成，PIT_LTMR64H 为高 32 位，用来显示定时器 1 的计数值，任意时刻 t_1 读取该寄存器，那么 PIT_LTMR64L 则显示此刻定时器 0 的值；PIT_LTMR64L 为低 32 位，只有先读取 PIT_LTMR64H，该寄存器才会更新定时器 0 的计数值，故该寄存器显示的是上一次读取 PIT_LTMR64H 时更新的定时器 0 的计数值。

2. PIT 驱动构件源码

1）基本编程步骤

使用 PIT 模块的某一通道进行定时，主要使用通道的 4 个寄存器。初始化 PIT 模块的某一通道的基本编程步骤如下。

（1）检查通道号，KL25 的通道号可以是 0 或 1；

（2）检查周期，即中断时间间隔，KL25 的系统总线时钟是 24MHz，由此计算周期的合力范围是 1～178 956，单位是 ms；

（3）打开 PIT 模块时钟源，配置 SIM_SCGC6 即可；

（4）使能 PIT 模块；

（5）配置选定通道的载入值寄存器、使能通道定时器及通道中断。

2）PIT 驱动构件源程序文件(pit.c)

```
//==========================================================
//文件名称：pit.c
//功能概要：KL25 pit 底层驱动程序文件
//版权所有：苏州大学恩智浦嵌入式中心(sumcu.suda.edu.cn)
//更新记录：2016-3-20    V4.0
//==========================================================
#include "pit.h"
//==========================================================
//函数名称：pit_init
//函数返回：无
//参数说明：channel:PIT 模块的通道号,0 或 1
//          freq:系统总线时钟频率,单位 kHz。例:系统总线时钟为 24MHz,则 freq=24 000。
//          int_ms:中断周期,以 ms 为单位。系统总线时钟为 24MHz,最大值为 178 956ms
//功能概要：PIT 模块初始化
//调用举例：pit_init(CH_0,PIT_WORK_FREQ,1000);即初始化 PIT 模块的 0 通道使用总线
//时钟频率,中断周期为 1s
//==========================================================
void pit_init(uint_8 channel,uint_16 freq,uint_32 int_ms)
{
    if(channel>1)
    {
        channel = 0;
    }
    if((int_ms<1)||(int_ms>178956))
        int_ms = 1000;
    BSET(SIM_SCGC6_PIT_SHIFT,SIM_SCGC6);              //开 PIT 时钟门
    BCLR(PIT_MCR_MDIS_SHIFT,PIT_MCR);                 //使能 PIT 模块
```

```
    BSET(PIT_MCR_FRZ_SHIFT,PIT_MCR);                    //调试模式下禁止
    PIT_LDVAL(channel) = int_ms * freq-1;
    PIT_TCTRL(channel) |= PIT_TCTRL_TEN_MASK;           //使能 PIT 通道定时器
    PIT_TCTRL(channel) |= (PIT_TCTRL_TIE_MASK);
}

//==========================================================================
//函数名称: pit_enable_int
//参数说明: channel:PIT 模块的通道号,0 或 1
//函数返回: 无
//功能概要: 使能某一通道的 PIT 中断
//调用举例: pit_enable_int(CH_0);使能 PIT 模块的通道 0 中断
//==========================================================================
void pit_enable_int(uint_8 channel)
{
    if(channel>1)
    {
        channel = 0;
    }
    PIT_TCTRL(channel)|=(PIT_TCTRL_TIE_MASK);           //开 PIT 通道中断

    NVIC_EnableIRQ(PIT_IRQn);                           //开 PIT 的 IRQ 中断
}

//==========================================================================
//函数名称: pit_disable_int
//参数说明: channel:PIT 模块的通道号,0 或 1
//函数返回: 无
//功能概要: 禁止某一通道的 PIT 中断
//调用举例: pit_disable_int(CH_0);禁止 PIT 模块的通道 0 中断
//==========================================================================
void pit_disable_int(uint_8 channel)
{
    if(channel>1)
    {
        channel = 0;
    }
    PIT_TCTRL(channel)&=~(PIT_TCTRL_TIE_MASK);          //关 PIT 通道中断

    NVIC_DisableIRQ(PIT_IRQn);                          //关 PIT 的 IRQ 中断
}
```

7.6　低功耗定时器 LPTMR 模块

7.6.1　低功耗定时器 LPTMR 模块功能概述

KL25 有一个低功耗定时器模块 LPTMR(Low Power Timer),可以被配置成具有可选预分频因子的定时器,也可以被配置成带有脉冲干扰滤波器功能的脉冲计数器。LPTMR

模块可以工作在所有的电源模式下，用以普通单次计时，时钟源多样，可以配置为 1kHz 的 LPO 时钟、32kHz 的 IRC 时钟、4MHz 的 IRC 时钟和 8MHz 的 OSCERCLK 时钟，并且时钟源分频范围大，最小分频为 1，最大分频 65 536。当采用 1kHz 的 LPO 时钟时，最大中断周期可达 50 天。LPTMR 正常工作电流可低至 777μA。

LPTMR 模块内含一个 16 位的递增定时器，对应的中断向量号为 44，非内核中断请求 IRQ 号为 28。在 LPTMR 模块初始化使能后，定时器从 0 开始加 1 计数，当计数器 CNR 的值与重载寄存器 LPTMR0_CMR 的值相等时就会置控制和状态寄存器 CSR 的比较标志位 TCF 并产生中断，程序转而运行该中断向量号对应的中断服务例程，并在例程中通过 w1c 来完成清比较标志位的工作，此时 LPTMR 模块就会重新开始计时。

7.6.2　LPTMR 驱动构件及使用方法

1. LPTMR 引脚

引脚 LPTMR0_ALT1、LPTMR0_ALT2 用于接入脉冲计数模式下的输入源，表 7-9 列出了 LPTMR 模块相关的引脚及其功能复用情况。

表 7-9　LPTMR 模块引脚

序号	引脚名	ALT0	ALT1	ALT2	ALT3	ALT4	ALT5	ALT6
41	PTA19	XTAL0	PTA19	—	UART1_TX	TPM_CLKIN1	—	LPTMR0_ALT1
62	PTC5/LLWU_P9	—	PTC5/LLWU_P9	SPI0_SCK	LPTMR0_ALT2	—	—	CMP0_OUT

2. LPTMR 驱动构件基本要素分析

LPTMR 模块的时钟源有很多，为区分不同时钟源，在头文件中给出了时钟源的宏定义。一般来说，在中断服务例程中，需要判断计数器是否产生溢出中断，因此，在头文件中给出了宏函数 LPTMR_GET_FLAG，用于判断计数器是否产生溢出中断。头文件中还给出了宏函数 LPTMR_CLEAR_FLAG，用于清中断标志。除宏定义及宏函数之外，头文件中还需给出对外接口函数：LPTMR 模块初始化函数 lptmr_init，形参为时钟源类别；使能中断函数 lptmr_enable_int 及禁止中断函数 lptmr_disable_int，无形参。这样可以满足 LPTMR 模块的基本编程。

3. LPTMR 构件头文件

```
//=============================================================
//文件名称：lptmr.h
//功能概要：lptmr 底层驱动构件头文件
//版权所有：苏州大学恩智浦嵌入式中心(sumcu.suda.edu.cn)
//更新记录：2016-3-20    V4.0
//=============================================================
#ifndef _LPTMR_H
#define _LPTMR_H
#include "common.h"
```

```
//LPTMR 时钟源
#define LPOCLK         1
#define IRC32KCLK      2
#define IRC4MCLK       3
#define OSCERCLK       4

//获取中断标志
#define LPTMR_GET_FLAG      ((LPTMR0_CSR & LPTMR_CSR_TCF_MASK) == \
                            LPTMR_CSR_TCF_MASK)
//清中断标志
#define LPTMR_CLEAR_FLAG      (LPTMR0_CSR |= LPTMR_CSR_TCF_MASK)

//=========================================
//函数名称: lptmr_init
//函数返回: 无
//参数说明: clktype,指明时钟源类别,参照 lptmr.h 中 LPTMR 时钟源的宏定义
//功能概要: LPTMR 模块初始化,配置 LPTMR 工作的时钟源,中断时间为 1s
//调用举例: lptmr_init(LPOCLK);配置 LPTMR 模块时钟源为 LPOCLK 时钟
//=========================================
void lptmr_init(uint_8 clktype);

//=========================================
//函数名称: lptmr_enable_int
//函数返回: 无
//参数说明: 无
//功能概要: 使能 LPTMR 模块中断
//调用举例: lptmr_enable_int();使能 LPTMR 模块中断
//=========================================
void lptmr_enable_int();

//=========================================
//函数名称: lptmr_disable_int
//函数返回: 无
//参数说明: 无
//功能概要: 禁止 LPTMR 模块中断
//调用举例: lptmr_disable_int();禁止 LPTMR 模块中断
//=========================================
void lptmr_disable_int();


#endif
```

4. LPTMR 驱动构件使用方法

设 LPTMR 模块使用内部 32.768kHz 的慢速时钟,中断周期初始化为 1000ms,使用步骤如下。

(1) 在 main()函数的初始化外设模块位置添加下列语句:

```
lptmr_init(IRC32KCLK);          //初始化 LPTMR
```

（2）在 main()函数的使能模块中断位置添加下列语句：

```
lptmr_enable_int();              //开 LPTMR 定时器中断
```

（3）在 isr.c 的中断服务例程 LPTMR0_IRQHandler 中进行计时。g_time 为记录时分秒的全局变量数组，每中断一次，秒值加 1。函数 SecAdd1 的实现见工程目录下的源文件 common.c。

```
//==============================================================
//函数名称：LPTMR0_IRQHandler
//参数说明：无
//函数返回：无
//功能概要：LPTMR0 中断服务例程。清中断标志，并完成计时，用于显示 MCU 运行时间
//调用举例：无
//==============================================================
void LPTMR0_IRQHandler(void)
{
    DISABLE_INTERRUPTS;                  //禁止总中断

    if(GET_LPTMR_FLAG)
    {
        SecAdd1(g_time);                 //g_time 是时分秒全局变量数组
        CLEAR_LPTMR_FLAG;                //清除 LPTMR 比较标志
    }

    ENABLE_INTERRUPTS;                   //开放总中断
}
```

5. LPTMR 驱动构件测试实例

LPTMR 构件的测试工程位于网上教学资源中的“..\program\CH07-KL25-LPTMR”文件夹。测试工程功能概述如下。

（1）串口通信格式：波特率 9600，1 位停止位，无校验。

（2）LPTMR 使用内部 32.768kHz 慢速时钟。

（3）上电或按复位按钮时，调试串口输出“苏州大学嵌入式实验室 LPTMR 构件测试用例!”。

（4）LPTMR 每 1s 产生一次中断，用于计时，调试串口每秒输出“MCU 记录的相对时间：00:00:01”，“00:00:01”为中断记录的时间，同时蓝色指示灯闪烁一次。

7.6.3 LPTMR 驱动构件的设计

1. LPTMR 模块的编程结构

KL25 的 LPTMR 模块共有 4 个 32 位寄存器，包括一个控制和状态寄存器(LPTMR0_CSR)，预分频寄存器(LPTMR0_PSR)，重载寄存器(LPTMR0_CMR)和计数器(LPTMR0_CNR)。通过对这些寄存器的编程，就可以使用 LPTMR 模块进行定时。有关 LPTMR 模块的存储器映像见 KL25 参考手册的 589 页。

1）控制和状态寄存器(LPTMRx_CSR)

LPTMRx_CSR结构如表7-10所示。

表7-10　LPTMRx_CSR结构

数据位	D7	D6	D5	D4	D3	D2	D1	D0
读	TCF	TIE	TPS		TPP	TFC	TMS	TEN
写	w1c							
复位	0							

D31～D8位为保留位段，不可写，复位值为0。

D7(TCF)——定时器比较标志。使能LPTMR后，当CNR等于CMR时，TCF将置位。禁用LPTMR或将TCF置位时，TCF清0。TCF=0，CNR值不等于CMR；TCF=1，CNR值等于CMR。

D6(TIE)——定时器中断使能。当TIE置位时，TCF置位并且产生LPTMR中断。TIE=0，禁用定时器中断；TIE=1，使能定时器中断。

D5～D4(TPS)——定时器引脚选择。配置脉冲计数模式下的输入源。只有禁用LPTMR，才能改变TPS。MCU类型不同，输入源的输入引脚也会不同。TPS=0，选择脉冲计数器输入0；TPS=1，选择脉冲计数器输入1；TPS=2，选择脉冲计数器输入2；TPS=3，选择脉冲计数器输入3。

D3(TPP)——定时器引脚极性。配置在脉冲计数器模式下的输入源的极性。只有禁用LPTMR，才能改变TPP。TPP=0，脉冲计数器的输入源是逻辑高电平，并且CNR将在上升沿增加；TPP=1，脉冲计数器的输入源是逻辑低电平，并且CNR将在下降沿增加。

D2(TFC)——定时器自由计数。当为0时，无论何时置位TCF都会使CNR复位。当为1时，CNR在定时器溢出时复位。只有禁用LPTMR，才能改变TFC。TFC=0；TCF置位时引起CNR复位；TFC=1，CNR在计数器溢出时复位。

D1(TMS)——定时器模式选择。配置LPTMR的工作模式。只有禁用LPTMR，才能改变TMS。TMS=0，普通计数器模式；TMS=1，脉冲计数器模式。

D0(TEN)——定时器使能。当清零TEN时，将重置LPTMR内部逻辑(包括CNR和TCF)。当TEN置位时，使能LPTMR，禁止更改CSR的D5～D1。TEN=0，禁用LPTMR，并复位内部逻辑；TEN=1，使能LPTMR。

2）预分频寄存器(LPTMRx_PSR)

D31～D8位为保留位段，不可写，复位值为0。

D6～D3(PRESCALE)——预分频值。配置在定时器普通计数器模式下预分频器的大小或脉冲计数器模式下脉冲干扰滤波器的宽度。只有禁用LPTMR，才能改变预分频值。如果预分频的值为n，则将预分频时钟进行2^{n+1}分频，输入引脚持续2^n个时钟上升沿之后脉冲干扰滤波器才会识别电平变化。

D2(PBYP)——绕过预分频器。当PBYP为1时，在定时器普通计数器模式或脉冲计数器模式选定的输入源直接给CNR提供时钟。当PBYP为0时，CNR由预分频器/脉冲干扰滤波器提供时钟。只有禁用LPTMR，才能改变PBYP。PBYP=0，使能预分频器/脉冲

干扰滤波器；PBYP=1，绕过预分频器/脉冲干扰滤波器。

D1、D0(PCS)——选择预分频器时钟。选择 LPTMR 预分频器/脉冲干扰滤波器的时钟。只有禁用 LPTMR，才能改变 PCS。PCS=0，选择预分频器/脉冲干扰滤波器时钟 0；PCS=1，选择预分频器/脉冲干扰滤波器时钟 1；PCS=2，选择预分频器/脉冲干扰滤波器时钟 2；PCS=3，选择预分频器/脉冲干扰滤波器时钟 3。

3）重载寄存器(LPTMRx_CMR)

D31～D16 位为保留位段，不可写，复位值为 0。

D15～D0(COMPARE)——比较值。使能 LPTMR 后，如果 CNR 等于 CMR 时，TCF 将置位。CMR 为 0 时，除非禁用 LPTMR，否则持续产生中断。使能 LPTMR 后，那么只有 TCF 置位，才能改变 CMR。

4）计数器(LPTMRx_CNR)

D31～D16 位为保留位段，不可写，复位值为 0。

D15～D0(COUNTER)——定时器计数值。复位值为 0，并且不可写仅可读。

2. LPTMR 构件源码

1）基本编程步骤

选定某一种时钟源，使用 LPTMR 进行计时，需使用 LPTMR 模块的编程结构介绍的 4 个寄存器。以选择外部 OSCERCLK 时钟为例，初始化 LPTMR 模块的基本编程步骤如下。

(1) 打开 LPTMR 模块时钟源，配置 SIM_SCGC5 即可；

(2) 选择外部 OSCERCLK 时钟；

(3) 配置预分频寄存器、重载寄存器、控制和状态寄存器，使能 LPTMR 模块。

2）LPTMR 驱动构件源程序文件(lptmr.c)

```
//==========================================================
//文件名称：lptmr.c
//功能概要：lptmr 底层驱动构件源文件
//版权所有：苏州大学恩智浦嵌入式中心(sumcu.suda.edu.cn)
//更新记录：2016-3-20    V4.0
//==========================================================
#include "lptmr.h"

//==========================================================
//函数名称：lptmr_init
//函数返回：无
//参数说明：clktype，指明时钟源类别，参照 lptmr.h 中 LPTMR 时钟源的宏定义
//功能概要：LPTMR 模块初始化，配置 LPTMR 工作的时钟源，中断时间为 1s
//调用举例：lptmr_init(LPOCLK)；配置 LPTMR 模块时钟源为 LPOCLK 时钟
//==========================================================
void lptmr_init(uint_8 clktype)
{
    uint_16 compare_value;

    SIM_SCGC5|=SIM_SCGC5_LPTMR_MASK;              //使能 LPTMR 模块时钟

    switch(clktype)
```

```
        {
            case LPOCLK:
                SIM_SOPT1 |= SIM_SOPT1_OSC32KSEL(0x01);    //选择 LPO 时钟
                LPTMR0_PSR |= LPTMR_PSR_PCS(1)|LPTMR_PSR_PBYP_MASK;
                compare_value=1000;
                break;
            case IRC32KCLK:
                MCG_C1 |= MCG_C1_IRCLKEN_MASK;             //使能内部参考时钟
                MCG_C2 |= MCG_C2_IRCS(0);  //MCG_C2[IRCS]=0,使能 32kHz 内部参考时钟

                LPTMR0_PSR=LPTMR_PSR_PCS(0)|LPTMR_PSR_PBYP_MASK;
                compare_value=32768;
                break;
            case IRC4MCLK:
                MCG_C1 |= MCG_C1_IRCLKEN_MASK;             //使能内部参考时钟
                MCG_C2 |= MCG_C2_IRCS(1);  //MCG_C2[IRCS]=1,使能 4MHz 内部参考时钟

                LPTMR0_PSR = LPTMR_PSR_PCS(0)|LPTMR_PSR_PRESCALE(0x7);
                                                                        //256 分频
                compare_value=15625;
                break;
            case OSCERCLK:                        //注意: 实验室 K64 评估板并没有该外部晶振
            default:
                //打开外部参考时钟
                SIM_SOPT1 |= SIM_SOPT1_OSC32KSEL(0x00);    //选择系统振荡器
                OSC0_CR |= OSC_CR_ERCLKEN_MASK;  //选择 EXTAL to drive  XOSCxERCLK

                LPTMR0_PSR = LPTMR_PSR_PCS(3)|LPTMR_PSR_PRESCALE(0x7);
                                                                        //256 分频
                compare_value=31250;
                break;
        }
        LPTMR0_CMR = LPTMR_CMR_COMPARE(compare_value);//设置比较寄存器值
        LPTMR0_CSR |= LPTMR_CSR_TEN_MASK;               //开启 LPTMR 模块设置
    }

    //==========================================================================
    //函数名称: lptmr_enable_int
    //函数返回: 无
    //参数说明: 无
    //功能概要: 使能 LPTMR 模块中断
    //调用举例: lptmr_enable_int();使能 LPTMR 模块中断
    //==========================================================================
    void lptmr_enable_int()
    {
        LPTMR0_CSR|=LPTMR_CSR_TIE_MASK;                 //开启 LPTMR 定时器中断
        NVIC_EnableIRQ(LPTMR0_IRQn);                    //开引脚的 IRQ 中断
    }
```

```
//===========================================================
//函数名称：lptmr_disable_int
//函数返回：无
//参数说明：无
//功能概要：禁止 LPTMR 模块中断
//调用举例：lptmr_disable_int();禁止 LPTMR 模块中断
//===========================================================
void lptmr_disable_int()
{
    LPTMR0_CSR &=~LPTMR_CSR_TIE_MASK;        //禁止 LPTMR 定时器中断
    NVIC_DisableIRQ(LPTMR0_IRQn);            //关引脚的 IRQ 中断
}
```

7.7　实时时钟 RTC 模块

7.7.1　RTC 模块功能概述

实时时钟(Real Time Clock,RTC)模块是一个独立供电的模块,在芯片掉电时由备用电源(V_{BAT})供电[①],确保 RTC 定时器正常运行,保持 RTC 寄存器状态。外部晶体振荡器为 RTC 定时器或其他外设提供 32.768kHz 的时钟;POR 块在 RTC 模块上电时产生一个上电复位信号,将所有的 RTC 寄存器初始化为默认状态;RTC 定时器由一个具有报警功能的 32 位秒计数寄存器和一个具有补偿功能的 16 位预分频寄存器组成;RTC 自身的软件复位控制位,也会初始化所有的 RTC 寄存器。注意,在 V_{BAT}掉电或 POR 中断时,不允许访问 RTC 的任何寄存器(除了控制寄存器),否则将产生总线错误。

RTC 正常情况下需要外接 32.768kHz 晶振、匹配电容、备用电源等元件,提供 MCU 掉电累计计时功能,但在 KL25 评估板上芯片主晶振与 RTC 晶振引脚因为复用在了一起产生冲突,无外部晶振,故无法提供掉电计时功能,RTC 可作普通累计计时和闹钟报警使用。在无法使用外部晶振的前提下,为保证 RTC 能够正常运行,时钟源一般选用 KL25 的 32.768kHz 的内部时钟,而使用该时钟需先将其输出到 CLKOUT 引脚(即 PTC3),然后通过导线连接到 RTC_CLKIN 引脚(即 PTC1),该时钟即可供 RTC 使用。闹钟报警通过一个专用的中断完成,来提醒 MCU 有没有发生报警事件。

KL25 有一个 RTC 模块,包含两个中断向量号,秒中断的中断向量号为 37,IRQ 号为 21,其他中断对应的中断向量号为 36,IRQ 号为 20。RTC 可工作在所有的低功耗模式下,并可输出 1Hz 的方波。RTC 模块的计数器为递增计数,累计计数时间可达 2 的 32 次方秒,超过 135 年。时钟源有多种,单次中断时间取决于时钟源的频率,当时钟源为 32.768kHz 时,中断时间为 1s,也是正常计时使用的中断时间,而如果中断为 1kHz 的 LPO 时钟,那么中断时间就变为 32.768s。环境温度为 25℃时,RTC 正常工作电流增加 357nA。

在 RTC 模块初始化使能后,每个时钟周期,预分频寄存器 TPR 的值加 1。每当该寄存

① KL25 芯片没有独立出来,有的芯片独立引出。

器的 D14 位由 1 变为 0 时,秒计数器 TSR 的值加 1,并与报警寄存器 TAR 的值进行比较,循环往复,并可产生秒中断,程序转而运行该中断向量号对应的中断服务例程,因为没有秒中断标志位故在例程中无须清除,在硬件设计上减少了软件开销;如果此时秒计数器 TSR 的值与报警寄存器 TAR 的值相等,就可产生报警中断,程序转而运行该中断向量号对应的中断服务例程,并在例程中通过重写报警寄存器 TAR 的值来完成清报警中断标志位 TAF 的工作。

7.7.2　RTC 驱动构件及使用方法

1. RTC 引脚

引脚 RTC_CLKOUT 用于给其他模块提供 1Hz 的 RTC 时钟,引脚 RTC_CLKIN 可用于给 RTC 模块提供工作时钟,表 7-11 列出了 RTC 模块相关的引脚及其复用情况。

表 7-11　RTC 引脚

序号	引脚名	ALT0	ALT1	ALT2	ALT3	ALT4
1	PTE0		PTE0		UART1_TX	RTC_CLKOUT
56	PTC1/LLWU_P6/RTC_CLKIN	ADC0_SE15/TSI0_CH14	PTC1/LLWU_P6/RTC_CLKIN	I2C1_SCL		TPM0_CH0

2. RTC 驱动构件基本要素分析

在中断服务例程中,需要判断计数器是否产生无效中断。因此,在头文件中给出了宏函数 RTC_GET_INVALID_FLAG,用于判断计数器是否产生无效中断。除无效中断外,还提供宏函数 RTC_GET_OVERFLOW_FLAG 用来判断溢出中断、宏函数 RTC_GET_ALAM_FLAG 用来判断报警中断。头文件中还给出了宏函数 RTC_CLEAR_FLAG,用于清中断标志。除宏函数外,头文件中还需给出对外接口函数:RTC 模块初始化函数 rtc_init,形参为秒计数器初始值及报警值;启动计时函数 rtc_start,无形参;停止计时函数 rtc_stop,无形参;复位报警值函数 rtc_reset_alarm_time,形参为报警值;复位秒计数器函数 rtc_reset_second_time,形参为秒计数值;使能中断函数 rtc_enable_int 及禁止中断函数 rtc_disable_int,无形参。这样可以满足 RTC 模块的基本编程。

3. RTC 驱动构件头文件

```
//===========================================================================
//文件名称:RTC.h
//功能概要:KL25 RTC 底层驱动程序头文件
//版权所有:苏州大学 NXP 嵌入式中心(sumcu.suda.edu.cn)
//更新记录:2016-3-20    V4.0
//===========================================================================
#ifndef _RTC_H
#define _RTC_H

//1 头文件
#include "common.h"
```

```
//2 宏定义
//获取中断标志
#define RTC_GET_INVALID_FLAG    ((RTC_SR & RTC_SR_TIF_MASK) == \
                                  RTC_SR_TIF_MASK)
#define RTC_GET_OVERFLOW_FLAG   ((RTC_SR & RTC_SR_TOF_MASK) == \
                                  RTC_SR_TOF_MASK)
#define RTC_GET_ALAM_FLAG   ((RTC_SR & RTC_SR_TAF_MASK) == \
                                  RTC_SR_TAF_MASK)
//清中断标志
#define RTC_CLEAR_FLAG    {RTC_SR &= (RTC_SR_TIF_MASK | RTC_SR_TOF_MASK | \
                    RTC_SR_TAF_MASK);RTC_TSR = 0;}

//===========================================================================
//函数名称: rtc_init
//函数参数: SecondTimes:秒计数器的初始值
//          AlarmTimes:报警寄存器的时间间隔
//函数返回: 无
//功能概要: RTC 驱动初始化。时钟源配置为 32.768kHz 的内部慢速时钟,需要通过导线把
//CLKOUT 的 PTC3 连到 RTC_CLKIN 的 PTC1
//调用举例: rtc_init(1,2);初始 RTC 的秒计数器为 1,报警值为 2
//===========================================================================
void rtc_init(uint32_t SecondTimes,uint32_t AlarmTimes);

//===========================================================================
//函数名称: rtc_start
//函数参数: 无
//函数返回: 无
//功能概要: 启动 RTC 模块计时
//调用举例: rtc_start();启动 RTC 模块计时
//===========================================================================
void rtc_start(void);

//===========================================================================
//函数名称: rtc_stop
//函数参数: 无
//函数返回: 无
//功能概要: 停止 RTC 模块计时
//调用举例: rtc_stop();关闭 RTC 模块计时
//===========================================================================
void rtc_stop(void);

//===========================================================================
//函数名称: rtc_reset_alarm_time
//函数参数: AlarmTimes 复位时赋给报警计时器的值
//函数返回: 无
//功能概要: 复位报警计时器
//调用举例: rtc_reset_alarm_time();写入 RTC 模块报警的新值
//===========================================================================
void rtc_reset_alarm_time(uint32_t AlarmTimes);
```

```
//==========================================================
//函数名称：rtc_reset_second_time
//函数参数：SecondTimes 复位时赋给秒计时器的值
//函数返回：无
//功能概要：复位秒计时器
//调用举例：rtc_reset_second_time();写入 RTC 模块计数器的新值
//==========================================================
void rtc_reset_second_time(uint32_t SecondTimes);

//==========================================================
//函数名称：rtc_enable_int
//函数参数：无
//函数返回：无
//功能概要：使能 RTC 模块中断
//调用举例：rtc_enable_int();使能 RTC 模块中断
//==========================================================
void rtc_enable_int();

//==========================================================
//函数名称：rtc_disable_int
//函数参数：无
//函数返回：无
//功能概要：禁止 RTC 模块中断
//调用举例：rtc_disable_int();禁止 RTC 模块中断
//==========================================================
void rtc_disable_int();

#endif
```

4. RTC驱动构件使用方法

设使用RTC模块进行秒计时，同时每秒产生一次报警中断，使用步骤如下。

(1) 在main()函数的初始化外设模块位置添加下列语句：

```
rtc_init(SecondTimes, AlarmTimes);          //RTC初始化
```

(2) 在main()函数的使能模块中断位置添加下列语句：

```
rtc_enable_int();                  //使能 RTC 模块中断
```

(3) 在main()函数的开总中断之前添加下列语句：

```
rtc_start();                        //启动 RTC 计时
```

(4) 在isr.c的中断服务例程RTC_IRQHandler中针对报警中断复位报警值，在中断服务例程RTC_Seconds_IRQHandler中进行秒计时。g_time为记录时分秒的全局变量数组，每中断一次，秒值加1。函数SecAdd1的实现见工程目录下的源文件common.c。

```
//======================================================================
//函数名称: RTC_IRQHandler
//函数返回: 无
//参数说明: 无
//功能概要: RTC 中断服务例程。如果是无效中断及溢出中断,清中断标志并复位
//计数器,如果是报警中断,复位报警值
//调用举例: 无
//======================================================================
void RTC_IRQHandler(void)
{
    if(RTC_GET_INVALID_FLAG)
    {
        printf("\r\n 进入 RTC 计时无效中断,原因: POR 或软件复位可导致 RTC 计数器
无效!");
        RTC_CLEAR_FLAG;
    }
    else if(RTC_GET_OVERFLOW_FLAG)
    {
        printf("\r\n 进入 RTC 计时溢出中断,原因: 秒计数器的值由 0xFFFFFFFF 加一!");
        RTC_CLEAR_FLAG;
    }
    else if(RTC_GET_ALAM_FLAG)
    {
        printf("\r\n 进入 RTC 报警中断,原因: 秒计数器的值等于报警值!");
        rtc_reset_alarm_time(AlarmTimes = AlarmTimes + 1);
    }
    else
    {
        printf("\r\nRTC 无中断事件发生,原因: POR 或软件复位后中断信号即有效!");
    }
}

//======================================================================
//函数名称: RTC_Seconds_IRQHandler
//函数返回: 无
//参数说明: 无
//功能概要: RTC 秒中断服务例程。完成累计计时,用于显示 MCU 运行时间。边沿敏感中断,
//每秒产生一次,无须状态位清零
//调用举例: 无
//======================================================================
void RTC_Seconds_IRQHandler(void)
{
    SecAdd1(g_time);          //g_time 是时分秒全局变量数组
}
```

5. RTC 驱动构件测试实例

RTC 构件的测试工程位于网上教学资源中的“..\program\CH07-KL25-RTC”文件夹。测试工程功能概述如下。

(1) 串口通信格式: 波特率 9600,1 位停止位,无校验。

(2) RTC 使用内部 32.768kHz 慢速时钟。

(3) 上电或按复位按钮时,调试串口输出"苏州大学嵌入式实验室 RTC 构件测试用例!"。

(4) RTC 每 1s 产生一次报警中断,调试串口输出"进入 RTC 报警中断,原因:秒计数器的值等于报警值!"。同时,RTC 每秒产生一次秒中断,用于计时,调试串口每秒输出"MCU 记录的相对时间:00:00:01","00:00:01"为秒中断记录的时间,同时蓝色指示灯闪烁一次。

7.7.3 RTC 驱动构件的设计

本节主要介绍如何根据 RTC 模块的各个寄存器的功能,结合上文给出的 rtc.h 编写具体的 RTC 的驱动。

1. RTC 模块的编程结构

KL25 的 RTC 模块共有 8 个 32 位寄存器,包括一个秒计数器(RTC_TSR),预分频寄存器(RTC_TPR),报警寄存器(RTC_TAR),补偿寄存器(RTC_TCR),控制寄存器(RTC_CR),状态寄存器(RTC_SR),保护寄存器(RTC_LR)和中断使能寄存器(RTC_IER)。通过对这些寄存器的编程,就可以使用 RTC 模块进行定时、报警。有关 RTC 模块的存储器映像见 KL25 参考手册的 598 页。

1) 控制寄存器(RTC_CR)

RTC_CR 结构见表 7-12。

表 7-12 RTC_CR 结构

高 16 位为保留位										
数据位	D13	D12	D11	D10	D9	D8	D3	D2	D1	D0
读/写	SC2P	SC4P	SC8P	SC16P	CLKO	OSCE	UM	SUP	WPE	SWR
复位	0									

D31~D14、D7~D4 位为保留位段,不可写,复位值为 0。

D13(SC2P)——晶振 2pF 电容配置。0 禁用 2pF 电容,1 使用 2pF 电容。

D12(SC4P)——晶振 4pF 电容配置。0 禁用 4pF 电容,1 使用 4pF 电容。

D11(SC8P)——晶振 8pF 电容配置。0 禁用 8pF 电容,1 使用 8pF 电容。

D10(SC16P)——晶振 16pF 电容配置。0 禁用 16pF 电容,1 使用 16pF 电容。

D9(CLKO)——时钟输出。0 表示 32kHz 时钟输出到其他外围设备,1 表示 32kHz 时钟不输出到其他外围设备。

D8(OSCE)——晶振使能。0 表示禁用 32.768kHz 振荡器;1 表示使能 32.768kHz 晶振。在置该位后,在等待晶振稳定后再使能定时器计时。

D3(UM)——更新模式,甚至状态寄存器处于保护时,允许 SR[TCE]被写。当置位时,如果置位 SR[TIF]或置位 SR[TOF]或者清零 SR[TCE],那么可以写 SR[TCE]。0 表示当保护时,寄存器不能写入;1 表示在有限制条件下保护时,寄存器可以写入。

D2(SUP)——监视器访问。0 表示不支持非监视器模式写访问并生成一个总线错误;1 表示支持非监视器模式写访问。

D1(WPE)——唤醒引脚使能。0 表示禁用 32.768kHz 振荡器；1 表示使能 32.768kHz 振荡器。在置该位后，在等待晶振稳定后再使能定时器计时。

D0(SWR)——软件复位。0 表示无软件复位；1 表示复位所有 RTC 寄存器(除了 SWR 位)。SWR 位可以 POR 清零或者软件清零。

2) 状态寄存器(RTC_SR)

D31～D5、D3 位为保留位段，不可写，复位值为 0。

D4(TCE)——定时器使能。当禁用定时器时，TPR 寄存器和 TSR 寄存器是可写的，但是定时器不会增加。当使能定时器后，TSR 寄存器和 TPR 寄存器不可写，但会增加。0 禁用定时器；1 使能定时器。

D2(TAF)——定时器报警标志。当 TAR[TAR]等于 TSR[TSR]时，警报标志置位。该位可以通过写 TAR 寄存器来清零。0 没有产生报警事件；1 产生报警事件。

D1(TOF)——定时器溢出标志。使能计数器后，当定时器溢出时，溢出标志置位。该位置位后，TSR 和 TPR 不会继续增加并且值为零。可通过写 TSR 寄存器清零。0 定时器没有溢出；1 定时器溢出并且值为 0。

D0(TIF)——定时器无效标志。定时器无效标志在 POR 或软件复位时会置位。该位置位后，TSR 和 TPR 不会继续增加并且值为零。可通过写 TSR 寄存器清零。0 定时器有效；1 定时器无效并且值为 0。

3) 其他寄存器

(1) 秒计数器(RTC_TSR)

D31～D0(TSR)——秒计数寄存器。使能定时器后，TSR 是只读的并且每秒加 1(前提是 SR[TOF]或 SR[TIF]没有置位)。当 SR[TOF]或 SR[TIF]置位时，定时器值为 0。禁用定时器后，TSR 可读可写，此时通过写 TSR 可清零 SR[TOF]或 SR[TIF]。支持 TSR 写 0，但是并不推荐这么做，因为当 SR[TIF]或 SR[TOF]置位(表示时间无效，TSR 是 0)时，TSR 读取值为 0。

(2) 预分频寄存器(RTC_TPR)

D31～D16 位为保留位段，不可写，复位值为 0。

D15～D0(TPR)——预分频器寄存器。使能定时器后，TPR 只读并且每个时钟周期加 1。当 SR[TOF]或 SR[TIF]置位时，定时器值为 0。禁用定时器后，TSR 可读可写。当 TPR 的 D14 位从逻辑 1 转换到逻辑 0 时，TSR[TSR]加 1。

(3) 报警寄存器(RTC_TAR)

D31～D0(TAR)——报警寄存器。使能定时器后，每当 TAR[TAR]等于 TSR[TSR]，SR[TAF]置位。写 TAR 可清零 SR[TAF]。

(4) 补偿寄存器(RTC_TCR)

D31～D24(CIC)——补偿间隔计数器。补偿间隔计数器的当前值。如果补偿间隔计数器值等于 0，那么它会加载 CIR 的值。如果 CIC 不等于 0，那么它每秒减 1。

D23～D16(TCV)——时间补偿值。当前值用在秒间隔的补偿逻辑中。如果 CIC 的值等于 0，那么每秒重新载入 TCR 的值。如果 CIC 不等于 0，那么它载入 0。

D15～D8(CIR)——补偿间隔寄存器，配置补偿间隔的秒数，可配置的秒数是 1～256。写入值比待补偿秒数小 1，例如，写 0 作为 1s 的补偿间隔。该寄存器是双缓冲的并且直到

当前补偿间隔结束后方才生效。

D7～D0(TCR)——时间补偿寄存器,在每秒内配置32.768kHz的时钟周期的数量。该寄存器是双缓冲的并且直到当前补偿间隔结束后方才生效。

80h 预分频寄存器每32 896个时钟周期溢出一次。

……

FFh 预分频器寄存器每32 769个时钟周期溢出一次。

00h 预分频器寄存器每32 768个时钟周期溢出一次。

01h 预分频器寄存器每32 767个时钟周期溢出一次。

……

7Fh 预分频器寄存器每32 641个时钟周期溢出一次。

(5) 保护寄存器(RTC_LR)

D31～D8位为保留位段,不可写,复位值为0。D7、D2～D0位为保留位段,不可写,复位值为1。

D6(LRL)——保护寄存器保护位。清零后,该位可通过POR或软件复位置位。LRL=0,保护寄存器被保护并且忽略写操作;LRL=1,保护寄存器不被保护并且写操作完全正常。

D5(SRL)——状态寄存器保护位。清零后,该位可通过POR或软件复位置位。SRL=0,保护状态寄存器并且忽略写操作;SRL=1,不保护状态寄存器并且写操作完全正常。

D4(CRL)——控制寄存器保护位。清零后,该位可通过POR或软件复位置位。CRL=0,保护控制寄存器并且忽略写操作;CRL=1,不保护控制寄存器并且写操作完全正常。

D3(TCL)——补偿寄存器保护位。清零后,该位可通过POR或软件复位置位。TCL=0,保护补偿寄存器并且忽略写操作;TCL=1,不保护补偿寄存器并且写操作完全正常。

(6) 中断使能寄存器(RTC_IER)

D31～D8为保留位段,不可写,复位值为0。D6～D5、D3位为保留位段,可读可写,复位值为0。

D7(WPON)——开启唤醒引脚。唤醒引脚是可选的,并非所有芯片都支持。每当唤醒引脚使能并且该位置位时,唤醒引脚将有效。WPON=0,无效;WPON=1,如果已使能唤醒引脚,那么唤醒引脚可用于唤醒芯片。

D4(TSIE)——定时器秒中断使能。秒中断是一个有专用中断向量号的边沿敏感的中断。每秒产生一次中断并且不需要软件开销(没有相应的中断状态标志需要清零)。TSIE=0,禁用秒中断;TSIE=1,使能秒中断。

D2(TAIE)——定时器报警中断使能。TAIE=0,定时器报警标志置位后并不产生一个中断;TAIE=1,定时器报警标志置位后产生一个中断。

D1(TOIE)——定时器溢出中断使能。TOIE=0,定时器溢出标志置位后并不产生一个中断;TOIE=1,定时器溢出标志置位后产生一个中断。

D0(TIIE)——定时器无效中断使能。TIIE=0,定时器无效标志置位后并不产生一个

中断；TIIE＝1，定时器无效标志置位后产生一个中断。

2. RTC 驱动构件源码

1）基本编程步骤

使用 RTC 模块进行定时和报警，主要使用除保护寄存器的另外 7 个寄存器。初始化 RTC 模块的基本编程步骤如下。

(1) 配置 32.768kHz 的 IRC 输出到芯片的引脚 CLKOUT；

(2) 使能引脚 RTC_CLKIN 作为 RTC 模块时钟输入的功能，并选择 RTC_CLKIN 作为 RTC 模块的时钟源，用于接入引脚 CLKOUT 的时钟信号；

(3) 打开 RTC 模块时钟源，配置 SIM_SCGC6 即可；

(4) 软件复位 RTC 模块，清除寄存器的写保护限制；

(5) 配置补偿寄存器和预分频寄存器，不对定时器进行补偿和分频；

(6) 配置秒计数器、报警寄存器的初始值；

(7) 使能 RTC 模块中断。

2）RTC 驱动构件源程序文件（rtc.c）

```
//===========================================================================
//文件名称：RTC.c
//功能概要：KL25 RTC 底层驱动程序源文件
//===========================================================================
#include "rtc.h"

//===========================================================================
//函数名称：rtc_clockconfig
//函数参数：clock，CLKOUT 引脚的时钟类型
//函数返回：无
//功能概要：rtc 时钟配置。将 clock 类型的时钟输出到 CLKOUT 引脚
//调用举例：rtc_clockconfig(CLKOUT_MCGIRCLK)；CLKOUT 引脚输出 IRC
//===========================================================================
void rtc_clockconfig(uint_8 clock)
{
    PORTC_PCR3 = PORT_PCR_MUX(0x5);
    SIM_SOPT2 |= SIM_SOPT2_CLKOUTSEL(clock); //复用 PTC3 为 CLKOUT，观测波形
    MCG_C1 |= MCG_C1_IRCLKEN_MASK|MCG_C1_IREFSTEN_MASK;
    MCG_C2 &= ~MCG_C2_IRCS_MASK;
}

//===========================================================================
//函数名称：rtc_init
//函数参数：SecondTimes:定时器秒寄存器的初始值
//          AlarmTimes:定时器报警寄存器的时间间隔
//函数返回：无
//功能概要：RTC 驱动初始化。时钟源配置为 32.768kHz 的内部慢速时钟，需要通过导线把
//CLKOUT 的 PTC3 连到 RTC_CLKIN 的 PTC1
//调用举例：rtc_init(1,2)；初始 RTC 的秒计数器为 1，报警值为 2
//===========================================================================
```

```
void rtc_init(uint32_t SecondTimes,uint32_t AlarmTimes)
{
    rtc_clockconfig(CLKOUT_MCGIRCLK);

    PORTC_PCR1 = PORT_PCR_MUX(0x1);          //使能 PTC1 为 RTC_CLKIN

    SIM_SCGC6 |= SIM_SCGC6_RTC_MASK;         //使能 RTC 时钟门控制

    RTC_CR= RTC_CR_SWR_MASK;        //软件复位 RTC 寄存器通过软件复位清除写保护
    RTC_CR  &= ~RTC_CR_SWR_MASK;             //软件复位之后清 SWR 位
    //如果定时器无效
    if (RTC_SR & RTC_SR_TIF_MASK) RTC_TSR = 0x00;
    //配置定时器补偿寄存器的时间间隔与时钟周期数
    RTC_TCR = RTC_TCR_CIR(0) | RTC_TCR_TCR(0);

    SIM_SOPT1 |= SIM_SOPT1_OSC32KSEL(0x02);  //选择 RTC_CLKIN 作为 RTC 时钟源

    RTC_TSR = SecondTimes;                   //初始化定时器秒寄存器

    RTC_TAR = AlarmTimes;                    //初始化定时器报警寄存器

    RTC_TPR = 0;                             //复位 RTC 定时器预分频器寄存器

    //使能秒中断,复位后 TAIE、TOIE、TIIE 值为 1
    RTC_IER |= RTC_IER_TSIE_MASK;

    rtc_stop();
}

//==========================================================
//函数名称: rtc_start
//函数参数: 无
//函数返回: 无
//功能概要: 启动 RTC 模块
//调用举例: rtc_start();启动 RTC 模块计时
//==========================================================
void rtc_start(void)
{
    RTC_SR |= RTC_SR_TCE_MASK;               //打开计数器
}

//==========================================================
//函数名称: rtc_stop
//函数参数: 无
//函数返回: 无
//功能概要: 关闭 RTC 模块
//调用举例: rtc_stop();关闭 RTC 模块计时
//==========================================================
void rtc_stop(void)
{
    RTC_SR &= ~RTC_SR_TCE_MASK;              //关闭计数器
```

```
}

//==========================================================================
//函数名称：rtc_reset_alarm_time
//函数参数：AlarmTimes 复位时赋给报警计时器的值
//函数返回：无
//功能概要：复位报警计时器
//调用举例：rtc_reset_alarm_time();写入 RTC 模块报警的新值
//==========================================================================
void rtc_reset_alarm_time(uint32_t AlarmTimes)
{
    RTC_TAR = AlarmTimes;
}

//==========================================================================
//函数名称：rtc_reset_second_time
//函数参数：SecondTimes 复位时赋给秒计时器的值
//函数返回：无
//功能概要：复位秒计时器
//调用举例：rtc_reset_second_time();写入 RTC 模块计数器的新值
//==========================================================================
void rtc_reset_second_time(uint32_t SecondTimes)
{
    RTC_TSR = SecondTimes;
}

//==========================================================================
//函数名称：rtc_enable_int
//函数参数：无
//函数返回：无
//功能概要：使能 RTC 模块中断
//调用举例：rtc_enable_int();使能 RTC 模块中断
//==========================================================================
void rtc_enable_int()
{
    //开 rtc 中断
    NVIC_EnableIRQ(RTC_IRQn);
    NVIC_EnableIRQ(RTC_Seconds_IRQn);
}

//==========================================================================
//函数名称：rtc_disable_int
//函数参数：无
//函数返回：无
//功能概要：禁止 RTC 模块中断
//调用举例：rtc_disable_int();禁止 RTC 模块中断
//==========================================================================
void rtc_disable_int()
{
    //关 rtc 中断
```

```
        NVIC_DisableIRQ(RTC_IRQn);
        NVIC_DisableIRQ(RTC_Seconds_IRQn);
    }
```

小　　结

本章介绍了ARM Cortex-M0+内核时钟SysTick的编程结构、构件设计及测试用例；介绍了脉宽调制、输入捕捉与输出比较的通用基础知识、TPM模块的驱动构件及使用方法、TPM模块的编程结构及驱动构件设计方法；分别给出了PIT、LPTMR、RTC的功能概述、构件及使用方法、构件的设计方法。

(1) ARM Cortex-M0+处理器内核中的SysTick模块作为硬件定时器，在嵌入式应用开发过程中，利用SysTick寄存器实现延时功能，可以节省CPU资源、化简嵌入式软件在Cortex-M内核芯片间的移植工作。

(2) PWM信号是一个高/低电平重复交替的输出信号，通常也叫脉宽调制波或PWM波。PWM信号的主要技术指标有周期、占空比、极性、脉冲宽度、分辨率、对齐方式等。PWM最常见的应用是电机控制。输入捕捉是用来监测外部开关量输入信号变化的时刻，这个时刻是定时器工作基础上的更精细时刻。输入捕捉的应用场合主要有测量脉冲信号的周期与波形。输出比较的功能是用程序的方法在规定的较精确时刻输出需要的电平，实现对外部电路的控制。输出比较的应用场合主要有产生一定间隔的脉冲。

(3) 定时器/脉宽调制模块(Timer/PWM Module，TPM)内含三个模块，分别称为TPM0、TPM1、TPM2，每个模块是独立的。TPM模块，除了作为基本定时器外，主要用于支持PWM、输入捕捉、输出比较功能。要求掌握TPM驱动构件基本定时、PWM、输入捕捉、输出比较函数的用法，了解TPM模块的编程结构及驱动构件设计方法。

(4) KL25/26内部有一个周期中断定时器模块PIT模块，内含两个通道，没有外部引脚。每个通道都有一个独立的32位的减1计数的计数器，时钟源固定为系统总线时钟并且不可分频。KL25/26内部有一个低功耗定时器模块LPTMR，可以被配置成具有可选预分频因子的定时器，也可以被配置成带有脉冲干扰滤波器功能的脉冲计数器。LPTMR模块可以工作在所有的电源模式下，用以普通单次计时，时钟源多样，且分频范围大，最小分频为1，最大分频为65 536。当采用1kHz的LPO时钟时，最大中断周期可达50天。LPTMR正常工作电流可低至777μA。KL25/26内部含有一个实时时钟RTC模块，是一个独立供电的模块，在芯片掉电时由备用电源(V_{BAT})供电，确保RTC定时器正常运行，保持RTC寄存器状态。

(5) 对于PIT、LPTMR、RTC，要求掌握其构件使用方法，了解其编程结构及驱动构件设计方法。

习　　题

1. 简述可编程定时器的主要思想。

2. 利用 SysTick 定时器延时，编写程序令三盏指示灯相隔延时 300ms 亮起。注意：后盏灯亮起时，前盏灯熄灭。

3. 分析当利用 SysTick 定时器设计的电子时钟，出现走快了或慢了的情况时，如何调整？

4. 给出 PWM 的基本含义及主要技术指标的含义。

5. 分别阐述 TPM 构件中 PWM、输入扑捉、输出比较的技术要点。

6. 分析归纳 PIT、LPTMR、RTC 各定时器模块的功能及应用场合，列表说明。

第 8 章　GPIO 应用——键盘、LED 及 LCD

本章导读：本章介绍嵌入式系统中常用的键盘、LED 数码管和 LCD 液晶显示，把它们作为 GPIO 的应用实例来看待，阐述它们的工作原理和编程方法。主要内容有：①键盘基础知识与键盘驱动构件设计；②LED 数码管基础知识与 LED 驱动构件设计；③LCD 基础知识与点阵字符型 LCD 驱动构件设计；④键盘、LED 及 LCD 驱动构件测试实例。本章提供的键盘、LED 和 LCD 驱动构件，可适用于不同型号 MCU，但需要注意硬件电路性能的差异。

本章参考资料：相关的 GPIO 的应用参考《KL 参考手册》第 10、11 章。

8.1　键盘基础知识与键盘驱动构件设计

本节在简要阐述键盘识别基本问题基础上，给出矩阵键盘的编程原理及键盘驱动构件的设计方法。

8.1.1　键盘模型及接口

键盘可由单个或多个按键组成，它是最简单的 MCU 数字量输入设备，通过键盘可输入数据或命令，从而实现简单的人机通信。

键盘的基本电路为接触开关，通、断两种状态分别用 0 和 1 表示。键盘电路模型以及其实际按键动作过程见图 8-1。MCU 通过检测与键盘相连接的 I/O 口的通断情况来确定键盘状态。

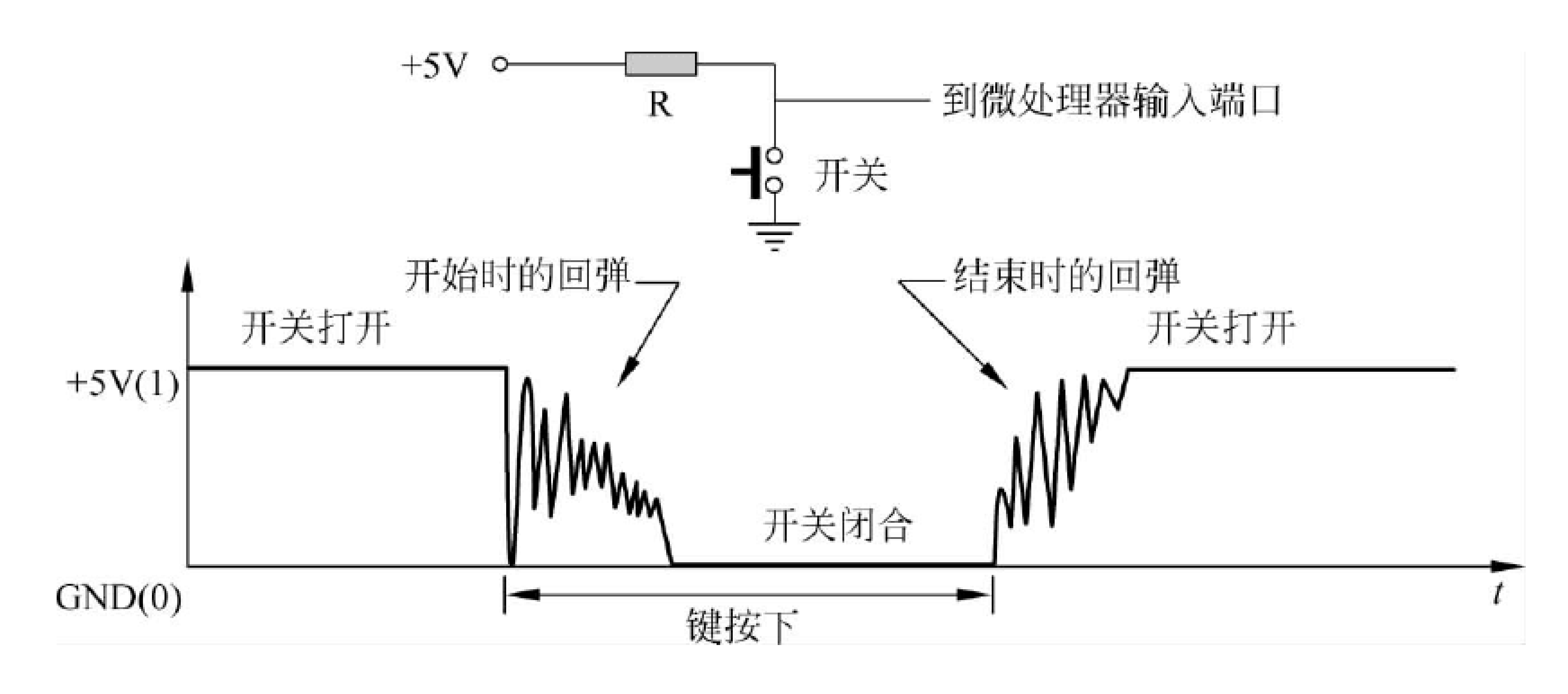


图 8-1　键盘模型及按键抖动示意图

键盘与 MCU 的连接方式主要有独立方式和矩阵方式。图 8-2 为独立方式示意图。

这种方式将每个独立按键按一对一的方式直接接到 MCU 的 GPIO 输入引脚。直接读取引脚状态，可确定哪个按键。这种方式实现简单，但占用 GPIO 引脚资源较多，一般只用

于按键数量少于 6 个的情况。

实际应用较多的是矩阵键盘。它由 m 条行线与 n 条列线组成。在行列线的每一个交点上设置一个按键。例如图 8-3，给出了一个 4×4 的矩阵键盘结构及实物图。8.1.2 节将讨论如何识别这种键盘的按键。

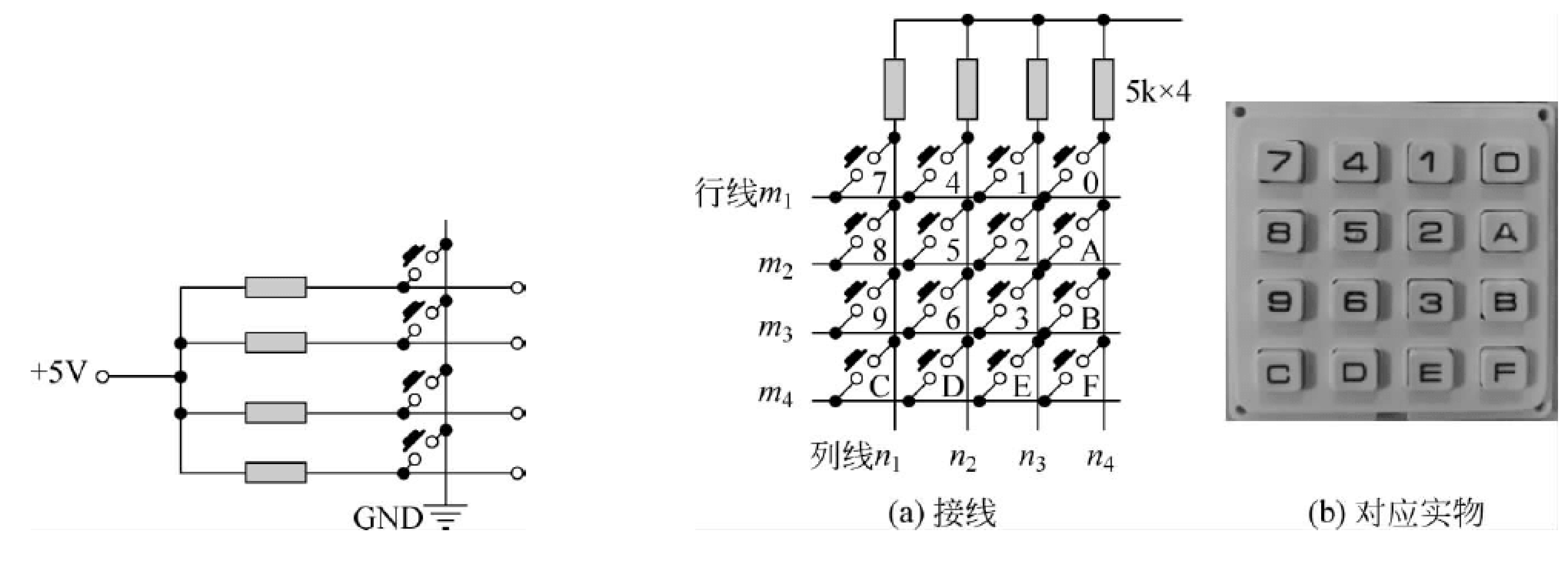


图 8-2　独立式键盘　　　　图 8-3　矩阵式键盘

8.1.2　键盘编程基本问题、扫描编程原理及键值计算

1. 键盘编程本问题

对于键盘编程应该了解几个问题：第一，如何识别键盘上的按键？第二，如何区分按键是否真正地被按下，还是抖动？第三，如何处理重键问题？

(1) 键的识别。如何知道键盘上哪个键被按下就是键的识别问题。若键盘上闭合键的识别由专用硬件实现，称为编码键盘；而靠软件实现的称为未编码键盘。在这里主要讨论未编码键盘的接口技术和键盘输入程序的设计。识别是否有键被按下，主要有查询法、定时扫描法与中断法等。而要识别键盘上哪个键被按下主要有行扫描法与行反转法。

(2) 抖动问题。当按键被按下时，会出现所按的键在闭合位置和断开位置之间跳几下才稳定到闭合状态的情况，当释放一个按键时也会出现类似的情况，这就是抖动问题。抖动持续的时间因操作者而异，一般为 5～10ms 之间，稳定闭合时间一般为十分之几秒到几秒，由操作者的按键动作所确定。在软件上，解决抖动的方法通常是延时等待抖动的消失或多次识别判定。

(3) 重键问题。所谓重键问题就是有两个及两个以上按键同时处于闭合状态的处理问题。在软件上，处理重键问题通常有连锁法与巡回法。

2. 行扫描法识别按键的基本原理

为了正确理解 MCU 键盘接口方法与编程技术，下面以 4×4 键盘为例说明行扫描法识别按键的基本编程原理。4×4 的键盘结构及实物如图 8-3 所示，图中列线（n_1～n_4）通过电阻接＋5V，当键盘上没有键闭合时，所有的行线和列线断开，列线 n_1～n_4 都呈高电平。当键盘上某一个键闭合时，则该键所对应的行线与列线短路。例如，图 8-3 中的标记为“6”的键被按下闭合时，行线 m_3 和列线 n_2 短路，此时 n_2 线上的电平由 m_3 的电位所决定。那么如何确定键盘上哪个按键被按下呢？**行扫描法识别按键基本原理就是，**把列线 n_1～n_4 接到 MCU 的输入引脚，行线 m_1～m_4 接到 MCU 的输出引脚，则在 MCU 的控制下，使行线 m_1

为低电平(0),其余三根行线 m_2、m_3、m_4 都为高电平(1),并读列线 n_1～n_4 状态。如果 n_1～n_4 都为高电平,则 m_1 这一行上没有键闭合,如果读出列线 n_1～n_4 的状态不全为高电平,那么为低电平的列线和 m_1 相交的键处于闭合状态;如果 m_1 这一行上没有键闭合,接着使行线 m_2 为低电平,其余行线为高电平,用同样方法检查 m_2 这一行上有无键闭合;以此类推,最后使行线 m_4 为低电平,其余的行线为高电平,检查 m_4 这一行上是否有键闭合。这种逐行逐列地检查键盘状态的过程称为对键盘的一次扫描。

MCU对键盘扫描可以采取程序控制的随机方式,空闲时扫描键盘。也可以采取定时控制,每隔一定时间,对键盘扫描一次。若接在键盘列线的MCU引脚具有下降沿或低电平中断功能,也可以采用中断方式,当键盘上有键闭合时,列线产生请求中断,CPU响应键盘输入中断,在中断服务例程中对键盘进行扫描,以识别哪一个键处于闭合状态。

3. 键值计算

键值是MCU获取硬件连接方式下每个按键的具有唯一性的数字表达。这里给出上述的4×4键盘的键值计算方法,以扫描方式获取键盘的输入值(键值)。按照上述接法,列线 n_1～n_4 接到MCU的输入引脚,行线 m_1～m_4 接到MCU的输出引脚。图8-3及表8-1中的实物图中的“7”“8”…“F”为键的“定义值”。行线的 m_1 和列线的 n_1 是对应着键盘上的“7”,按扫描法原理,所以当“7”键被按下时,m_1 和 n_1 这两条线是低电平,取0,其余位为1。使用排序 $m_4m_3m_2m_1n_4n_3n_2n_1$ 表达键值,可放在一个字节内。这样,定义值“7”对应的键值为二进制11101110,即十六进制0xEE,同理定义值“8”对应的键值是二进制11011110,即十六进制0xED,等等。由此可得到键盘定义值与键值的对应关系,见表8-1。

表8-1　键盘的定义值及键值

列 / 行	n_1		n_2		n_3		n_4	
	定义值	键值	定义值	键值	定义值	键值	定义值	键值
m_1	“7”	0xEE	“4”	0xED	“1”	0xEB	“0”	0xE7
m_2	“8”	0xDE	“5”	0xDD	“2”	0xDB	“A”	0xD7
m_3	“9”	0xBE	“6”	0xBD	“3”	0xBB	“B”	0xB7
m_4	“C”	0x7E	“D”	0x7D	“E”	0x7B	“F”	0x77

如果是一个 M 行 N 列的矩阵键盘,其键值可用二进制数为 $m_Mm_{M-1}\cdots m_1n_Nn_{N-1}\cdots n_1$ 表达,根据前面的分析可知第 i 行第 j 列的键值应该是第 $i+N$ 位和第 j 位为0。超过8位,但小于16位,可以用16位无符号数字表达。

8.1.3　键盘驱动构件的设计

1. 键盘驱动构件要素分析

关于键盘的硬件接线。使用宏定义描述硬件接线,且每个接线单独宏定义,更具普适性,这样,若键盘接在MCU的不同引脚,只需修改键盘的硬件接线宏定义即可。关于键盘消抖问题,可以采用多次扫描的方式消除键盘按下或弹开时产生的抖动。关于键值与按键的对应,在kb.c文件的头部给出,用户可查阅使用或根据实际按键修改。

2. 键盘构件头文件

```
//======================================================================
// 文件名称：kb.h
// 功能概要：键盘构件头文件
// 版权所有：苏州大学 NXP 嵌入式中心(sumcu.suda.edu.cn)
// 版本更新：2012-06-08 V1.0，2016-05-12    V6.0(WYH)
//======================================================================

#ifndef _KB_H                           //防止重复定义(_KB_H 开头)
#define _KB_H

#include "common.h"                     //包含公共要素头文件
#include "gpio.h"                       //包含 gpio 头文件

//键盘(KB)硬件接线
#define m1   (PTC_NUM | 11)             //4 根行线硬件连接
#define m2   (PTC_NUM | 10)
#define m3   (PTC_NUM | 9)
#define m4   (PTC_NUM | 8)
#define n1  (PTC_NUM | 7)               //4 根列线硬件连接
#define n2  (PTC_NUM | 6)
#define n3  (PTC_NUM | 5)
#define n4  (PTC_NUM | 4)

//====================接口函数声明====================
//======================================================================
//函数名称：KBInit
//函数返回：无
//参数说明：无
//功能概要：初始化键盘模块
//======================================================================
void KBInit(void);

//======================================================================
//函数名称：KBScanN
//函数返回：键值，无键按下返回 0xFF
//参数说明：重复扫描键盘的的次数(scan_cnt)，建议 1～20 之间
//功能概要：多次扫描键盘，返回键值，scan_cnt 小于等于 1，直接返回扫描一次键值，否则再
//          继续扫描到(scan_cnt×0.5)次相同的键值，且键值不为 0xFF 时返回该键值，
//          否则返回最后一次扫描值
//======================================================================
uint_8 KBScanN(uint_8 KB_count);

//======================================================================
//函数名称：KBDef
//函数返回：无
//参数说明：键值 valve
//功能概要：键值转为定义值函数
//======================================================================
uint_8 KBDef(uint_8 valve);

#endif                                  //防止重复定义(结尾)
```

3. 键盘构件源文件

本程序的关键是根据行扫描法识别按键基本原理，在理解键值计算方法基础上，理解扫描一次键盘获得键值的函数 KBScan1()。

```
//==============================================================
// 文件名称: kb.c
// 功能概要: 键盘构件源文件
//==============================================================
#include "kb.h"

//键盘键值与定义值对应表
const uint_8 KBtable[] =
{
    0xEE,'7',  0xED,'4',  0xEB,'1',  0xE7,'0',
    0xDE,'8',  0xDD,'5',  0xDB,'2',  0xD7,'A',
    0xBE,'9',  0xBD,'6',  0xBB,'3',  0xB7,'B',
    0x7E,'C',  0x7D,'D',  0x7B,'E',  0x77,'F',
    0x00
};

//说明: 用一个字节表达键值,其位顺序是{m4,m3,m2,m1,n4,n3,n2,n1}
uint_16 kbm[4]={m1, m2, m3, m4};
uint_16 kbn[4]={n1, n2, n3, n4};

//内部函数声明
uint_8 KBScan1(void);

//==============================================================
//函数名称: KBInit
//函数返回: 无
//参数说明: 无
//功能概要: 初始化键盘模块
//==============================================================
void KBInit(void)
{
    uint_8 i;
    //定义列线为输入,且上拉
    for(i = 0;i < 4;i++)
    {
        gpio_init(kbn[i],GPIO_IN,0);
        gpio_pull(kbn[i],1);
    }
    //定义行线为输出,且初始状态为低电平,低电平是为中断方式时产生下降沿做准备
    for(i = 0;i < 4;i++)
    {
        gpio_init(kbm[i], GPIO_OUTPUT, 0);
    }
}
```

```
//======================================================
//函数名称: KBScanN
//函数返回: 键值
//参数说明: 重复扫描键盘的的次数(scan_cnt),建议 1~20 之间
//功能概要: 多次扫描键盘,返回键值,scan_cnt 小于等于 1,直接返回扫描一次键值,否则再
//          扫描到连续(scan_cnt×0.5)次相同的键值,且键值不为 0xFF 时返回该键值,
//          否则返回 0xFF
//======================================================
uint_8 KBScanN(uint_8 scan_cnt)
{
    uint_8 i,KB_value,KB_value_last,same_count;
    same_count=0;                                        //键值相同次数变量=0
    KB_value_last=0xFF;
    if(scan_cnt<1 || scan_cnt>20)
        scan_cnt=1;
    //以下多次扫描消除抖动
    for (i=1; i<=scan_cnt; i++)
    {
        KB_value = KBScan1();                            //扫描一次,获取一次键值
        if ((KB_value == KB_value_last) && (KB_value!=0xFF))//相等情况
        {
            same_count++;                                //键值相同次数变量+1
            if(same_count>=scan_cnt * 0.5)   break;      //返回键值
        }
        else                                             //不相等情况
        {
            KB_value_last=KB_value;                      //保存当前键值
            same_count=1;                                //键值相同次数变量=1
        }
    }
    return  KB_value;                                    //返回键值
}

//======================================================
//函数名称: KBDef
//函数返回: 无
//参数说明: 键值 value
//功能概要: 键值转为定义值函数
//======================================================
uint_8 KBDef(uint_8 value)
{
    uint_8 KeyPress;                       //键定义值
    uint_8 i;
    i = 0;
    KeyPress = 0xff;
    while (KBtable[i] != 0x00)             //在键盘定义表中搜索欲转换的键值,直至表尾
    {
        if(KBtable[i] == value)            //在表中找到相应的键值
        {
```

```
                KeyPress = KBtable[i+1];          //取出对应的键定义值
                break;
            }
            i += 2;                               //指向下一个键值,继续判断
        }
        return KeyPress;
}

//-----------------------------以下为内部函数存放处-----------------------------------
//==================================================================
//函数名称:KBScan1
//函数返回:扫描到的键值
//参数说明:无
//功能概要:扫描一次 4×4 键盘,返回扫描到的键值,若无按键,返回 0xff
//==================================================================
uint_8 KBScan1(void)
{
    uint_8 keyvalue;                              //声明键值临时变量
    uint_8 i,n,flag;                              //声明临时变量

    keyvalue = 0xff;                              //键值临时变量初值
    flag=0;

    KBInit();                                     //键盘初始化
    //进行行扫描
    for (i = 0; i <= 3; i++)
    {
        //令第 i 行=低,其余各行拉高
        for(n=0;n<4;n++) { gpio_set(kbm[n],1);}
        gpio_set(kbm[i],0);
        //延时
        asm("NOP");
        asm("NOP");
        //检查列线,看是否有由于按键被按下而被拉低的列
        for(n=0;n<4;n++)
        {
            if(0==(gpio_get(kbn[n])))             //找到具体列线
            {
                BCLR(i+4,keyvalue);               //计算键值(对应行线=0)
                BCLR(n,keyvalue);                 //计算键值(对应列线=0)
                //至此,有按键,且键值在临时变量 keyvalue 中
                flag=1;                           //有按键标志
                break;
            }
        }
    if (1==flag) break;                           //有按键
    }
    return(keyvalue);                             //返回键值。(若无按键,该值为 0xff)
}
//-------------------------------------内部函数结束--------------------------------
```

8.2　LED 数码管基础知识与 LED 驱动构件设计

由 8 个发光二极管(Light Emitting Diode,LED)按照组成数字 0～9 的方式进行物理连接,形成了 LED 数码管,也可简称 LED。本节在介绍 8 段 LED 数码管显示原理的基础上,给出了 LED 数码管驱动构件设计。

8.2.1　LED 数码管基础知识

对于 LED 编程需要了解下列几个问题:第一,所用 LED 是几段(一个发光二极管形成一段),共阴极还是共阳极?第二,所选 LED 的电气参数怎样?如额定功率、额定电流是多少?如果对上述两个问题有明确的了解,那么对 LED 编程和封装 LED 构件就变得容易多了。

LED 的选择需要根据实际应用需求来决定,若只需要显示数字"0"～"9",则只需 7 段 LED 就够了,若同时又要显示小数点,则需使用 8 段 LED。8 段数码管由 8 个发光二极管 LED 组成。

1. 单个 LED 数码管工作原理

MCU 是通过 I/O 脚来控制 LED 某段发光二极管的亮暗从而达到显示某个数字的目的。那么怎样才能使 LED 发光二极管亮暗呢?首先应了解所选用的是共阴极数码管还是共阳极数码管。若为共阴数码管,则公共端需要接地,若为共阳则公共端接电源正极(参见图 8-4),数码管外形如图 8-5 所示。图中标记为 a、b、c、d、e、f、g、h 的被称为一个"段",即一个发光二极管。共阴极 8 段数码管的信号端高电平有效,只要在各段加上高电平信号即可使相应的段发光,比如要使 a 段发光,则在 a 段加上高电平即可。共阳极的 8 段数码管则相反,在相应的段加上低电平即可使该段发光。因而一个 8 段数码管就必须有 8 位(即 1 个字节)数据来控制各个段的亮暗。比如对共阳极 8 段数码管,[hgfedcba]=[01111111]时,h 段亮;当[hgfedcba]=[10000000]时,除 h 段外,其他段均亮。到此基本弄清对一个 LED 编程的原理了,下面需要注意的是在进行硬件连接时需要注意所选用的 LED 的电气参数,如能承受的最大电流、额定电压。根据其电气参数来选择使用限流电阻或电流放大电路。

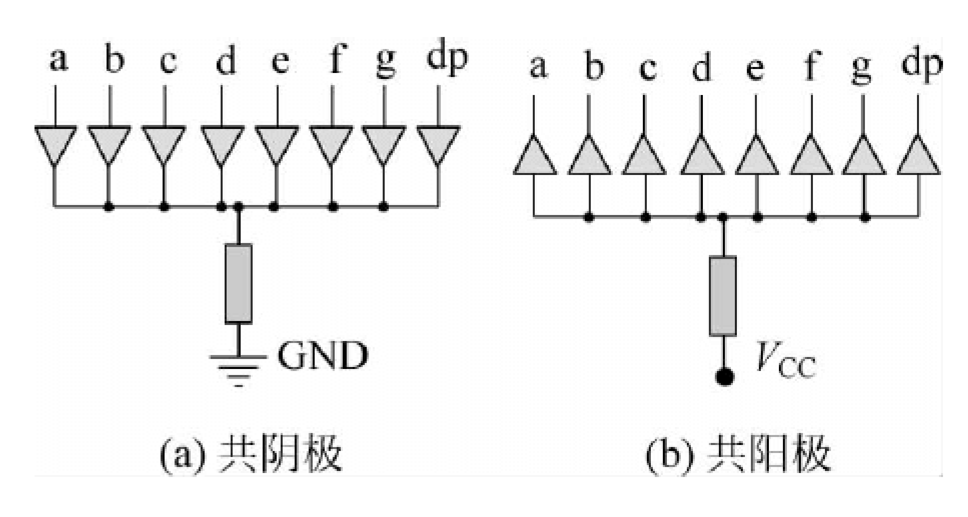


图 8-4　数码管

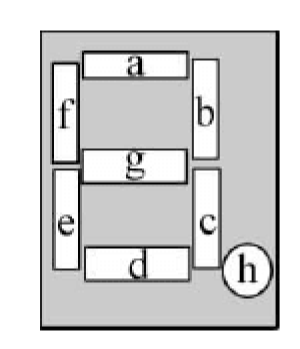


图 8-5　数码管外形

2. 多个 LED 数码管工作原理

实际应用中,大多数情况是多个数码管。下面介绍如何对多个 LED 数码管进行编程。

那么是不是如前面所述一样,有几个8段数码管,就必须有几个字节的数据线来控制各个数码管的亮暗呢?这样控制虽然简单,却不切实际,MCU也不可能提供这么多的引脚用来控制数码管。为此往往是通过一个称为数据口的8位数据线来控制段。而8段数码管的公共端,原来接到固定的电平(对共阴极是GND,对共阳极是Vcc),现在接MCU的一个输出引脚,由MCU来控制,通常叫"位选信号",而把这些由 n 个数码管合在一起的数码管组称为**n 连排数码管**。这样,MCU的12根引脚就可控制如图8-6所示的一个4连排的数码管。若是要控制更多的数码管,还可以考虑外加一个译码芯片。图8-6是一个4连排的共阴极数码管,各个数码管的段信号端(称为数据端)分别对应相连,可以由MCU的8个引脚控制,同时还有4个位选信号(称为控制端,这里的"位"就是位置的意思,位选就是指向第几个数码管之意),用于分别选中要显示数据的数码管,可用MCU的4个引脚来控制。每个时刻只让一个数码管有效(即只有一个位选信号为0,其他为1),由于人眼的"视觉暂留"(约100ms左右)效应,看起来则是同时显示的效果。这种 n 连排数码管也称动态扫描数码管,其含义就是任何一个时刻,只有一个数码管显示,而整体上看起来一起显示,是由于MCU对其动态刷新,而人眼具有"视觉暂留"效应而造成的现象。

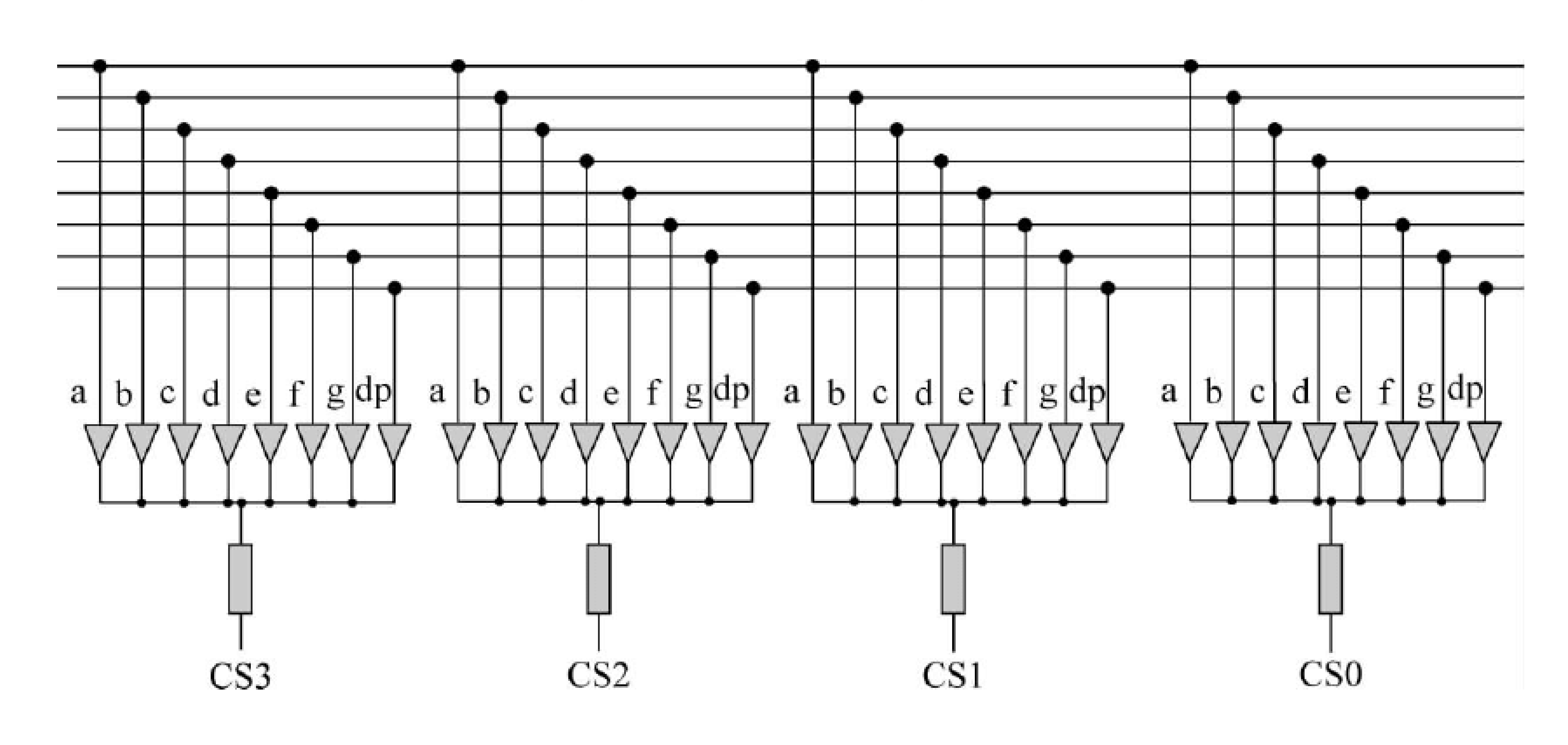


图8-6　4连排共阴极8段数码管

8.2.2　LED驱动构件设计及使用方法

以下给出4连排LED驱动构件设计。LED驱动构件属于应用构件,因为它需调用基础底层驱动GPIO构件,设计好的LED驱动构件头文件led.h及源代码文件led.c放在工程框架的"..\06_App_Component\led"文件夹中。硬件连接举例见图8-7,图中右下角为驱动三极管电路。

1. LED驱动构件要素分析

关于LED的硬件接线。使用宏定义描述硬件接线,且每个接线单独宏定义,更具普适性,这样,若LED接在MCU的不同引脚,只需修改LED的硬件接线宏定义即可。关于位选问题。虽然一个时刻只能显示一个数码管,但可以使用静态变量确定下次要显示的位选信号,这样LEDshow函数就可使用4字节数组作为形参,实际调用时,将待显示的4字节数组作为实参传入即可。每隔10ms左右,在定时中断服务例程中,调用该函数一次,由于人眼的"视觉暂留",可稳定地显示需要的数字。关于显示码。在led.c文件的头部给出,供查阅使用。

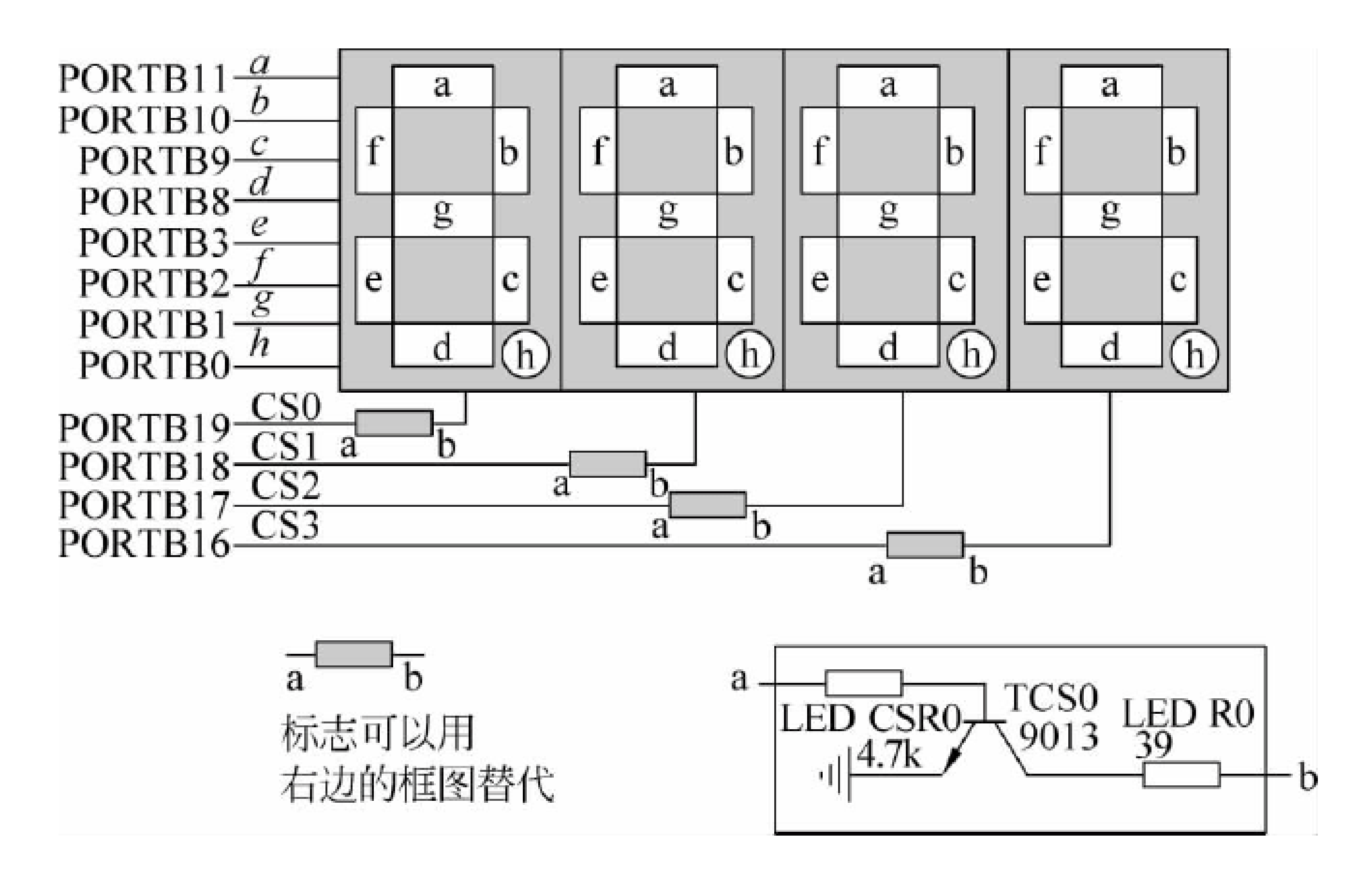


图 8-7　MCU 与 4 连排 8 段数码管的连接

2. LED 驱动构件头文件

```
//=============================================
// 文件名称：led.h
// 功能概要：led 构件头文件
// 版权所有：苏州大学嵌入式中心(sumcu.suda.edu.cn)
// 版本更新：2011-03-17　V1.0；2016-05-02　V6.0(WYH)
//=============================================

#ifndef _LED_H                              //防止重复定义(开头)
#define _LED_H

#include "common.h"                         //包含公共要素头文件
#include "gpio.h"                           //包含 gpio 头文件

//LED 的硬件接线
#define LED_D1    (PTB_NUM|11)              //LED 数据口
#define LED_D2    (PTB_NUM|10)
#define LED_D3    (PTB_NUM|9)
#define LED_D4    (PTB_NUM|8)
#define LED_D5    (PTB_NUM|3)
#define LED_D6    (PTB_NUM|2)
#define LED_D7    (PTB_NUM|1)
#define LED_D8    (PTB_NUM|0)
#define LED_CS0   (PTB_NUM|19)              //LED 位选口
#define LED_CS1   (PTB_NUM|18)
#define LED_CS2   (PTB_NUM|17)
#define LED_CS3   (PTB_NUM|16)

//显示码表(见 led.c 文件)--0～9 之外的序号需要参考此表

//=============================================
//函数名称：LEDInit
//函数返回：无
```

```
//参数说明：无
//功能概要：LED 初始化。即从 MCU 角度,定义所有线输出,并给出初始值一律为 0(全暗)
//=========================================================
void LEDInit();

//=========================================================
//函数名称：LEDshow
//函数返回：无
//参数说明：data[4]：显示的内容。可显示的数字 0~9,0.~9.,E,F,全亮,全暗(见显示码表)
//功能概要：将数组 data 内容显示在 LED 上。本函数调用一次会显示数组中的一个字符,
//        因此需延时 10ms 左右调用一次本函数才能将数组内容全部显示在 LED 上
//=========================================================
void LEDshow(uint_8 data[4]);

#endif    //防止重复定义(结尾)
```

3. LED 驱动构件源程序文件中的码表

```
//    ——
//    |   |
//    ——
//    |   |
//    —— .
//上为 8 段数码管的示意图,最顶端编号为 a 段,顺时针旋转下来分别为 bcdef 段,最中心
//的为 g 段,右下角的. 为 h 段,数码管显示码为[hgfedcba]形式的一个字节数据。
//下面的数组为共阴极接法下的显示码表,举例说明计算方式：如需要显示数字 8,则应
//当 h 段暗,其他段亮,即 h 段为 0,其他为 1,二进制为[01111111],十六进制为 0x7F
//若为共阳极接法,各段值刚好与共阴相反,若需显示 8,h 段应为 1,其他为 0,二进制
//形式为[10000000],十六进制形式为 0x80;其他显示数字计算方式与此相同。
//显示码表
const uint_8 LEDcodetable[24] =
//   0    1    2    3    4     5    6    7    8    9
  {0x3F,0x06,0x5B,0x4F,0x66,  0x6D,0x7D,0x07,0x7F,0x6F,
//   10   11   12   13   14     15   16   17   18   19  此行为数组下标值
//   0.   1.   2.   3.   4.     5.   6.   7.   8.   9.
   0xBF,0x86,0xDB,0x4F,0x66,  0x6D,0x7D,0x07,0xFF,0x6F,
//   20   21   22      23  此行为数组下标值
//   E    F   (全亮)  (全暗)
   0x79,0x71,  0xFF,   0x00};
```

4. LED 驱动构件的使用方法

1) 先导工作——在 led.h 硬件接线

根据 LED 实际使用的 MCU 引脚,修改 led.h 文件中“LED 的硬件接线”,例如：

```
//LED 的硬件接线
#define LED_D1    (PTB_NUM|11)        //LED 数据口
#define LED_D2    (PTB_NUM|10)
...
```

2）在“includes.h”文件中声明全局变量位置声明 LED 显示缓冲区数组
例如，LED 显示缓冲区数组名为 g_LEDBuffer，则：

```
uint_8   g_LEDBuffer[4];      //LED 显示缓冲区
```

3）在“main.c”文件中“变量赋初值”位置给 LED 显示缓冲区赋初值
设初始显示“0235”，则：

```
//LED 缓冲区赋值
g_LEDBuffer[0]=0;
g_LEDBuffer[1]=2;
g_LEDBuffer[2]=3;
g_LEDBuffer[3]=5;
```

4）在“isr.c”的某一定时中断处理函数中添加调用 LEDshow 函数即可
如在 SysTick_Handler 中，添加：

```
//LED 显示
LEDshow(g_LEDBuffer);
```

这样只要 main 函数中正常初始化并开启 SysTick 中断及总中断，LED 就正常显示了。任何程序中改变 LED 显示缓冲区 g_LEDBuffer 的值，LED 显示随即改变！见网上教学资源本章测试例程。

5. LED 驱动构件源文件

```
//=========================================================================
// 文件名称：led.c
// 功能概要：led 构件源文件
//=========================================================================
#include "led.h"

(显示码表，前面已经给出，这里略)

//led 位选端口
const uint_16 led_cs[4]=
{
    LED_CS0,LED_CS1,LED_CS2,LED_CS3
};

//led 数据端口
const uint_16 led_d[8]=
{
    LED_D1,LED_D2,LED_D3,LED_D4,LED_D5,LED_D6,LED_D7,LED_D8
};

//=========================================================================
//函数名称：LEDInit
//函数返回：无
```

```
//参数说明：无
//功能概要：LED 初始化。即从 MCU 角度,定义所有线输出,并给出初始值一律为 0(全暗)
//=====================================================================
void LEDInit()
{
    uint_8 i = 0;
    //定义 8 根数据线为输出,初始输出 0
    for(i = 0;i < 8;i++)   gpio_init(led_d[i], 1, 0);
    //定义 4 位选线为输出,初始输出 0
    for(i = 0;i < 4;i++)   gpio_init(led_cs[i], 1, 0);
}

//=====================================================================
//函数名称：LEDshow
//函数返回：无
//参数说明：data[4]：显示的内容。可显示的数字 0～9,0.～9.,E,F,全亮,全暗(见显示码表)
//功能概要：将数组 data 内容显示在 LED 上。本函数调用一次会显示数组中的一个字符,
//          因此需延时 10ms 左右调用一次本函数才能将数组内容全部显示在 LED 上
//=====================================================================
void LEDshow(uint_8 data[4])
{
    static   uint_8   LEDi=0;             //声明静态变量(位选线索引变量)并赋初值 0
    uint_8   i,j,m,n;
    //(1)取待显示数组的第 LEDi 字节的数据,并转为显示码,赋给 m
    j=data[LEDi];                         //取待显示数字或序号
    m=LEDcodetable[j];                    //使用头文件中常数表,将数字或序号转为显示码
    //(2)在 LED 的第 LEDi 位置显示 m
    for (i=0;i<=3;i++) gpio_set(led_cs[i], 0);   //位选全部置 0(全暗)
    for (i=0;i<=7;i++)                    //一个字节数据 m 上线
    {
        n = (m>>i) & 0x01;                //获得 m 的一位
        gpio_set(led_d[i], n);            //一位数据上线
    }
    gpio_set(led_cs[LEDi], 1);            //选择的位选线置 1(一个字节显示出来了)
    //(3)位选线索引变量 LEDi 下移一位,并设定 LEDi 界限
    LEDi++;                               //位选线索引变量+1,下次调用,将显示下一个字符
    if (LEDi>=4)   LEDi=0;                //大于 4 位选线索引变量置 0,从头开始
}
```

8.3　LCD 基础知识与 LCD 驱动构件设计

本节简要概述液晶显示器(Liquid Crystal Display，LCD)的基本特点及分类方法。给出点阵字符型液晶显示模块的驱动构件设计实例。

8.3.1 LCD 的特点和分类

LCD 作为电子信息产品的主要显示器件，相对于其他类型的显示部件来说，有其自身的特点，概要如下。

(1) 低电压微功耗。LCD 的工作电压一般为 3～5V，每平方厘米的液晶显示屏的工作电流为 μA 级，所以液晶显示器件为电池供电的电子设备的首选显示器件。

(2) 平板型结构。LCD 的基本结构是由两片玻璃组成的很薄的盒子。这种结构具有使用方便、生产工艺简单等优点。特别是在生产上，适宜采用集成化生产工艺，通过自动生产流水线可以快速、大批量地生产。

(3) 使用寿命长。LCD 器件本身几乎没有劣化问题。若能注意器件防潮、防压、防止划伤、防止紫外线照射、防静电等，同时注意使用温度，LCD 可以使用很长时间。

(4) 被动显示。对 LCD 来说，环境光线越强显示内容越清晰。人眼所感受的外部信息 90%以上是外部物体对光的反射，而不是物体本身发光，所以被动显示更适合人的视觉习惯，更不容易引起疲劳。这在信息量大、显示密度高、观看时间长的场合显得更重要。

(5) 显示信息量大且易于彩色化。LCD 与 CRT 相比，由于 LCD 没有荫罩限制，像素可以做得很小，这对于高清晰电视是一种理想的选择方案。同时液晶易于彩色化，方法也很多。特别是液晶的彩色可以做得更逼真。

(6) 无电磁辐射。CRT 工作时，不仅会产生 X 射线，还会产生其他电磁辐射，影响环境。LCD 则不会有这类问题。

液晶显示器件分类方法有多种，这里简要介绍以下几种分类方法。

1. 按电光效应分类

所谓电光效应是指在电的作用下，液晶分子的初始排列改变为其他排列形式，从而使液晶盒的光学性质发生变化，也就是说以电通过液晶分子对光进行了调制。不同的电光效应可以制成不同类型的显示器件。

按电光效应分类，LCD 可分为电场效应类、电流效应类、电热写入效应类和热效应类。其中，电场效应类又可分为扭曲向列效应(TN)类、宾主效应(GH)类和超扭曲效应(STN)类等。MCU 系统中应用较广泛的是 TN 型和 STN 型液晶器件，由于 STN 型液晶器件具有视角宽、对比度好等优点，几乎所有 32 路以上的点阵 LCD 都采用了 STN 效应结构，STN 型正逐步代替 TN 型而成为主流。

2. 按显示内容分类

按显示内容分类，LCD 可分为字段型(或称为笔画型)、点阵字符型、点阵图形型三种。

字段型 LCD 是指以长条笔画状显示像素组成的液晶显示器件。字段型 LCD 以七段显示最常用，也包括为专用液晶显示器设计的固定图形及少量汉字。字段型 LCD 主要应用于数字仪表、计算器、计数器中。

点阵字符型 LCD 是指显示的基本单元由一定数量的点阵组成，专门用于显示数字、字母、常用图形符号及少量自定义符号或汉字。这类显示器把 LCD 控制器、点阵驱动器、字符存储器等全做在一块印刷电路板上，构成便于应用的液晶显示模块。点阵字符型液晶显示模块在国际上已经规范化，有统一的引脚与编程结构。点阵字符型液晶显示模块有内置 192 个字符，另外用户可自定义 5×7 点阵字符或 5×11 点阵字符若干个。显示行数一般为

1行、2行、4行三种。每行可显示8个、16个、20个、24个、32个、40个字符不等。

点阵图形型除了可显示字符外,还可以显示各种图形信息、汉字等,显示自由度大。常见的模块点阵从80×32到640×480不等。

3. 按LCD的采光方式分类

LCD器件按其采光方式分类,分为带背光源与不带背光源两大类。不带背光的LCD显示是靠背面的反射膜将射入的自然光从下面反射出来完成的。大部分计数、计时、仪表、计算器等计量显示部件都是用自然光源,可以选择使用不带背光的LCD器件。如果产品需要在弱光或黑暗条件下使用,可以选择带背光型LCD,但背光源增加了功耗。

8.3.2 点阵字符型LCD模块控制器HD44780

点阵字符型LCD专门用于显示数字、字母、图形符号及少量自定义符号。这类显示器把LCD控制器、点阵驱动器、字符存储器、显示体及少量的阻容元件等集成为一个液晶显示模块。鉴于字符型液晶显示模块目前在国际上已经规范化,其电特性及接口特性是统一的,因此只要设计出一种型号的接口电路,在指令上稍加修改即可使用各种规格的字符型液晶显示模块。

这里以日立公司(HITACHI)生产的HD44780点阵字符型LCD模块控制器为例,阐述其编程基本方法。兼容型号主要有SED1278(SEIKO EPSON)、KS0066(SAMSUNG)、NJU6408(NER JAPAN RADIO)等。主要应用特点如下。

(1) 单+5V电源供电(宽温型需要加−7V驱动电源)。

(2) 使用MCU的GPIO可编程控制LCD。

(3) 液晶显示屏是以若干5×8或5×11点阵块组成的显示字符群。每个点阵块为一个字符位,字符间距和行距都为一个点的宽度。

(4) 内部具有字符发生器ROM(Character-Generator ROM,CG ROM),可显示192种字符(160个5×7点阵字符和32个5×10点阵字符)。

(5) 具有64字节的自定义字符RAM(Character-Generator RAM,CG RAM),可以定义8个5×8点阵字符或4个5×11点阵字符。

(6) 具有64字节的数据显示RAM(Data-Display RAM,DD RAM),供显示编程时使用。

1. HD44780的引脚信号

HD44780的外部接口信号线一般有14条,有的型号显示器使用16条,其中与MCU的接口有8条数据线、3条控制线,见表8-2。

表8-2　HD44780的引脚信号

引脚号	符号	电平	方向	引脚含义说明
1	Vss			电源地
2	Vdd			电源(+5V)
3	V0			液晶驱动电源(0～5V)
4	RS	H/L	输入	寄存器选择:1-数据寄存器 0-指令寄存器
5	R/$\overline{W}$	H/L	输入	读写操作选择:1-读操作 0-写操作

续表

引脚号	符号	电平	方向	引脚含义说明
6	E	H/L H→L	输入	使能信号：R/$\overline{W}$ =0,E 下降沿有效 R/$\overline{W}$=1,E=1 有效
7～10	DB0～DB3		三态	8 位数据总线的低 4 位,若与 MCU 进行 4 位传送时,此 4 位不用
11～14	DB4～DB7		三态	8 位数据总线的高 4 位,若与 MCU 进行 4 位传送时,只用此 4 位
15、16	E1、E2		输入	上下两行使能信号,只用于一些特殊型号

2．HD44780 的时序信号

图 8-8 给出了 HD44780 的写操作时序,图 8-9 给出了 HD44780 的读操作时序。

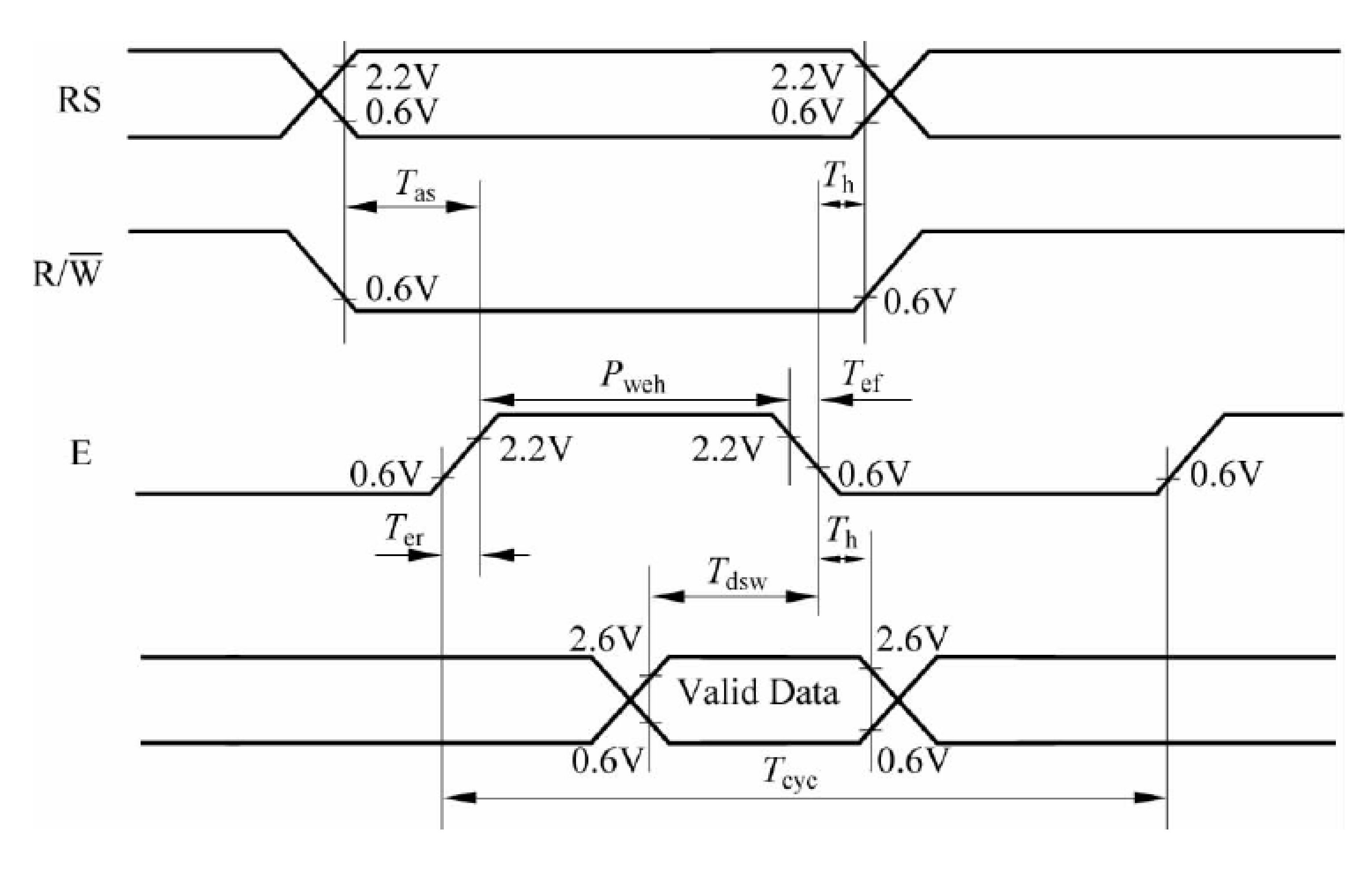


图 8-8　HD44780 的写操作时序

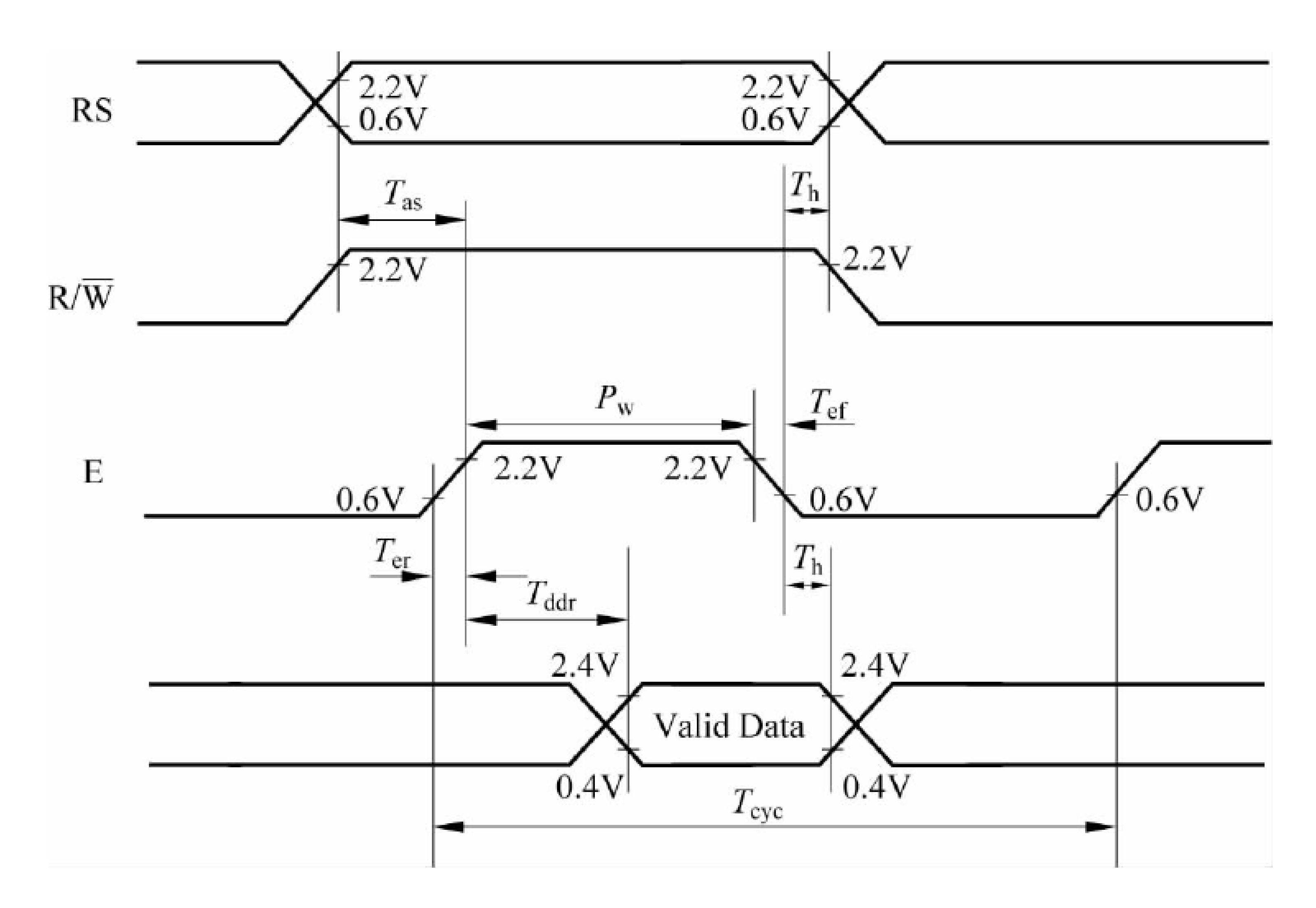


图 8-9　HD44780 的读操作时序

3．HD44780 的编程结构

从编程角度看,HD44780 内部主要由指令寄存器(IR)、数据寄存器(DR)、忙标志

(BF)、地址计数器(AC)、显示数据寄存器(DD RAM)、字符发生器ROM(CG ROM)、字符发生器RAM(CG RAM)及时序发生电路构成。

1）指令寄存器(IR)

IR用于MCU向HD44780写入指令码。IR只能写入，不能读出。当RS=0、R/$\overline{W}$=0时，数据线DB7～DB0上的数据写入指令寄存器IR。

2）数据寄存器(DR)

DR用于寄存数据。当RS=1、R/$\overline{W}$=0时，数据线DB7～DB0上的数据写入数据寄存器DR，同时DR的数据由内部操作自动写入DD RAM或CG RAM。当RS=1、R/$\overline{W}$=1时，内部操作将DD RAM或CG RAM送到DR中，通过DR送到数据总线DB7～DB0上。

3）忙标志(BF)

令RS=0、R/$\overline{W}$=1，在E信号高电平的作用下，BF输出到总线的DB7上，MCU可以读出判别。BF=1，表示组件正在进行内部操作，不能接收外部指令或数据。

4）地址计数器(AC)

AC作为DD RAM或CG RAM的地址指针。如果地址码随指令写入IR，则IR的地址码部分自动装入地址计数器AC之中，同时选择了相应的DD RAM或CG RAM单元。

AC具有自动加1或自动减1功能。当数据从DR送到DD RAM(或CG RAM)，AC自动加1。当数据从DD RAM(或CG RAM)送到DR，AC自动减1。当RS=0、R/$\overline{W}$=1时，在E信号高电平的作用下，AC所指向的内容送到DB7～DB0。

5）显示数据寄存器(DD RAM)

DD RAM用于存储显示数据，共有80个字符码。对于不同的显示行数及每行字符个数，所使用的地址不同，例如：

8×1(8个字符，一行)

字符位置	1	2	3	4	5	6	7	8
地　　址	00	01	02	03	04	05	06	07

16×1(16个字符，一行)

字符位置	1	2	…	8	9	10	…	16
地　　址	00	01	…	07	08	09	…	0F

16×2(每行16个字符，共两行)

字符位置	1	2	…	8	9	10	…	16
第一行地址	00	01	…	07	08	09	…	0F
第二行地址	40	41	…	47	48	49	…	4F

16×4(每行16个字符，共4行)

字符位置	1	2	…	8	9	10	…	16
第一行地址	00	01	…	07	08	09	…	0F
第二行地址	40	41	…	47	48	49	…	4F
第三行地址	10	11	…	17	18	19	…	1F
第四行地址	50	51	…	57	58	59	…	5F

具体的对应关系，可参阅使用说明书。

6）字符发生器 ROM(CG ROM)

CG ROM 由 8 位字符码生成 5×7 点阵字符 160 种和 5×10 点阵字符 32 种，见图 8-10。图中给出了 8 位字符编码与字符的对应关系，可以直接使用。其中大部分与 ASCII 码兼容。

Upper 4 Bits / Lower 4 Bits	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
xxxx0000	CG RAM (1)			0	@	P	`	p				ー	タ	ミ	α	p
xxxx0001	(2)		!	1	A	Q	a	q			。	ア	チ	ム	ä	q
xxxx0010	(3)		"	2	B	R	b	r			「	イ	ツ	メ	β	θ
xxxx0011	(4)		#	3	C	S	c	s			」	ウ	テ	モ	ε	∞
xxxx0100	(5)		$	4	D	T	d	t			、	エ	ト	ヤ	μ	Ω
xxxx0101	(6)		%	5	E	U	e	u			・	オ	ナ	ユ	σ	ü
xxxx0110	(7)		&	6	F	V	f	v			ヲ	カ	ニ	ヨ	ρ	Σ
xxxx0111	(8)		'	7	G	W	g	w			ァ	キ	ヌ	ラ	g	π
xxxx1000	(1)		(	8	H	X	h	x			ィ	ク	ネ	リ	√	x̄
xxxx1001	(2)		)	9	I	Y	i	y			ゥ	ケ	ノ	ル	⁻¹	y
xxxx1010	(3)		*	:	J	Z	j	z			ェ	コ	ハ	レ	j	千
xxxx1011	(4)		+	;	K	[	k	{			ォ	サ	ヒ	ロ	ˣ	万
xxxx1100	(5)		,	<	L	¥	l	|			ャ	シ	フ	ワ	¢	円
xxxx1101	(6)		-	=	M	]	m	}			ュ	ス	ヘ	ン	£	÷
xxxx1110	(7)		.	>	N	^	n	→			ョ	セ	ホ	゛	ñ	
xxxx1111	(8)		/	?	O	_	o	←			ッ	ソ	マ	゜	ö	█

图 8-10　HD44780 内藏字符集

7）字符发生器 RAM(CG RAM)

CG RAM 是提供给用户自定义特殊字符用的，它的容量仅为 64B，编址为 00～3FH。作为字符字模使用的仅是一个字节中的低 5 位，每个字节的高三位留给用户作为数据存储器使用。如果用户自定义字符由 5×7 点阵构成，可定义 8 个字符。

4. HD44780 的指令集

1）清屏(Clear Display)

RS、R/$\overline{W}$=00，DATA=0000 0001。清屏指令使 DD RAM 的内容全部被清除，屏幕光标回原位，地址计数器 AC=0。运行时间(250kHz)约为 1.64ms。

2）归位(Return Home)

RS、R/$\overline{W}$=00，DATA=0000 001＊(注：“＊”表示任意，下同)。归位指令使光标和光标所在位的字符回原点(屏幕的左上角)。但 DD RAM 单元内容不变。地址计数器 AC=0。运行时间(250kHz)约为 1.64ms。

3）输入方式设置(Entry Mode Set)

RS、R/$\overline{W}$=00，DATA=0000 01AS。该指令设置光标、画面的移动方式。下面解释 A、

S位的含义。A=1时数据读写操作后，AC自动增1；A=0时数据读写操作后，AC自动减1。若S=1，当数据写入DD RAM显示将全部左移(A=1)或全部右移(A=0)，此时光标看上去未动，仅仅是显示内容移动，但从DD RAM中读取数据时，显示不移动；S=0时显示不移动，光标左移(A=1)或右移(A=0)。

4) 显示开关控制(Display ON/OFF Control)

RS、R/$\overline{W}$=00，DATA=0000 1DCB。该指令设置显示、光标及闪烁开、关。D：显示控制，D=1，开显示(Display ON)；D=0，关显示(Display OFF)。C：光标控制，C=1，开光标显示；C=0，关光标显示。B：闪烁控制，B=1，光标所指的字符同光标一起以0.4s交变闪烁；B=0，不闪烁。运行时间(250kHz)约为40μs。

5) 光标或画面移位(Cursor or Display Shift)

RS、R/$\overline{W}$=00，DATA=0001 S/C R/L ** 。该指令使光标或画面在没有对DD RAM进行读写操作时被左移或右移，不影响DD RAM。S/C=0、R/L=0，光标左移一个字符位，AC自动减1；S/C=0、R/L=1，光标右移一个字符位，AC自动加1；S/C=1、R/L=0，光标和画面一起左移一个字符位；S/C=1、R/L=1，光标和画面一起右移一个字符位。运行时间(250kHz)约为40μs。

6) 功能设置(Function Set)

RS、R/$\overline{W}$=00，DATA=001 DL NF ** 。该指令为工作方式设置命令(初始化命令)。对HD44780初始化时，需要设置数据接口位数(4位或8位)、显示行数、点阵模式(5×7或5×10)。DL：设置数据接口位数，DL=1，8位数据总线DB7～DB0；DL=0，4位数据总线DB7～DB4，而DB3～DB0不用，在此方式下数据操作需两次完成。N：设置显示行数，N=1，两行显示；N=0，一行显示。F：设置点阵模式，F=0，5×7点阵；F=1，5×10点阵。运行时间(250kHz)约为40μs。

7) CG RAM地址设置(CG RAM Address Set)

RS、R/$\overline{W}$=00，DATA=01 A5 A4 A3 A2 A1 A0。该指令设置CG RAM地址指针。A5～A0=00 0000～11 1111。地址码A5～A0被送入AC中，在此后，就可以将用户自定义的显示字符数据写入CG RAM或从CG RAM中读出。运行时间(250kHz)约为40μs。

8) DD RAM地址设置(DD RAM Address Set)

RS、R/$\overline{W}$=00，DATA=1 A6 A5 A4 A3 A2 A1 A0。该指令设置DD RAM地址指针。若是一行显示，地址码A6～A0=00～4FH有效；若是两行显示，首行址码A6～A0=00～27H有效，次行址码A6～A0=40～67H有效。在此后，就可以将显示字符码写入DD RAM或从DD RAM中读出。运行时间(250kHz)约为40μs。

9) 读忙标志BF和AC值(Read Busy Flag and Address Count)

RS、R/$\overline{W}$=01，DATA=BF AC6 AC5 AC4 AC3 AC2 AC1 AC0。该指令读取BF及AC。BF为内部操作忙标志，BF=1，忙；BF=0，不忙。AC6～AC0为地址计数器AC的值。当BF=0时，送到DB6～DB0的数据(AC6～AC0)有效。

10) 写数据到DDRAM或CGRAM(Write Data to DDRAM or CG RAM)

RS、R/$\overline{W}$=10，DATA=实际数据。该指令根据最近设置的地址，将数据写入DD RAM或CG RAM中。实际上，数据被直接写入DR，再由内部操作写入地址指针所指的DD RAM或CG RAM。运行时间(250kHz)约为40μs。

11）读 DDRAM 或 CGRAM 数据(Read Data from DDRAM or CGRAM)

RS、R/$\overline{W}$=11,DATA=实际数据。该指令根据最近设置的地址,从 DD RAM 或 CG RAM 读数据到总线 DB7～DB0 上。运行时间(250kHz)约为 40μs。

8.3.3　LCD 构件设计

本节给出点阵字符型 LCD 的一个编程实例。

1. LCD 驱动构件要素分析

关于 LCD 的硬件接线,使用宏定义描述硬件接线,且每个接线单独宏定义,更具普适性,这样,若 LCD 接在 MCU 的不同引脚,只需修改 LCD 的硬件接线宏定义即可。下面的 lcd.h 文件中给出一例。本书例程的 LCD,可以显示 32 个字符,其编码见图 8-10,大部分编码符合 ASCII 码编码规范。LCDShow(uint_8 data[32])的形参就是 32 个字符,编程时,可以先声明 32 个字节的数组作为显示缓冲区,作为实参传给 LCDShow,即可在 LCD 上显示该内容。

2. LCD 构件头文件

```
//========================================================================
// 文件名称: lcd.h
// 功能概要: lcd 构件头文件
// 版权所有: 苏州大学嵌入式中心(sumcu.suda.edu.cn)
// 版本更新: 2011-12-06   V1.0;   2016-05-02   V6.0(WYH)
//========================================================================

#ifndef _LCD_H                          //防止重复定义(_LCD_H 开头)
#define _LCD_H

#include "common.h"                     //包含公共要素头文件
#include "gpio.h"                       //包含 gpio 头文件

//LCD 硬件接线
#define LCD_RS   (PTD_NUM|7)            //LCD 寄存器选择信号引脚
#define LCD_RW   (PTD_NUM|6)            //LCD 读写信号引脚
#define LCD_E    (PTD_NUM|5)            //LCD 读写信号引脚
#define LCD_D0   (PTD_NUM|4)            //LCD 数据引脚
#define LCD_D1   (PTD_NUM|3)
#define LCD_D2   (PTD_NUM|2)
#define LCD_D3   (PTD_NUM|1)
#define LCD_D4   (PTD_NUM|0)
#define LCD_D5   (PTC_NUM|17)
#define LCD_D6   (PTC_NUM|16)
#define LCD_D7   (PTC_NUM|13)

//========================================================================
//函数名称: LCDInit
//函数返回: 无
//参数说明: 无
//功能概要: LCD 初始化
```

```
//==========================================================================
void LCDInit();

//==========================================================================
//函数名称：LCDShow
//函数返回：无
//参数说明：data[32]：需要显示的数组
//功能概要：通过液晶显示 data 中的 32 字节数据
//==========================================================================
void LCDShow(uint_8 data[32]);

#endif  //防止重复定义(结尾)
```

3．LCD 驱动构件的使用方法

1）先导工作——在 lcd.h 硬件接线

根据 LCD 实际使用的 MCU 引脚，修改 lcd.h 文件中“LCD 的硬件接线”，例如：

```
#define LCD_RS    (PTD_NUM|7)          //LCD 寄存器选择信号引脚
#define LCD_RW    (PTD_NUM|6)          //LCD 读写信号引脚
...
```

2）在“includes.h”文件中声明全局变量位置声明 LCD 显示缓冲区数组

例如，LCD 显示缓冲区数组名为 g_LCDBuffer，则：

```
uint_8   g_LCDBuffer[32];     //LCD 显示缓冲区
```

3）在“main.c”文件中“初始化外设模块”位置对 LCD 进行初值化

```
LCDInit();                         //LCD 初始化//lcd 显示初始字符
```

4）在可需要 LCD 显示的地方调用 LCDshow 函数

只要对 g_LCDBuffer[]赋可以显示的 ASCII 码(图 8-10 中编码)，调用 LCDshow 函数，即可在 LCD 屏幕上显示 g_LCDBuffer[]中的内容。

```
LCDShow(g_LCDBuffer);
```

4．LCD 构源文件(lcd.c)

理解本程序的关键在于根据图 8-8 和图 8-9 的 HD44780 的时序，完成 LCDCommand (uint_8 RS，uint_8 cmdordata)。这是 MCU 如何向 LCD 中的 HD44780 控制器送出一个字节的命令或数据的函数。为了使程序的可复用性面提高，程序中使用指令延时的地方，适当加大，这样使得即使用于总线时钟频率较高的 MCU，延时时间也能达到要求。

```
//==========================================================================
// 文件名称：lcd.c
```

```
// 功能概要：lcd 构件头文件
//======================================================================
#include "lcd.h"

//8 个数据位引脚,使用数组,方便循环编程
static uint_16 LCD_D[8]=
{
    LCD_D0,LCD_D1,LCD_D2,LCD_D3,LCD_D4,LCD_D5,LCD_D6,LCD_D7
};

//内部函数原型声明
void LCDCommand(uint_8 RS,uint_8 cmdordata);

//======================================================================
//函数名称：LCDInit
//函数返回：无
//参数说明：无
//功能概要：LCD 初始化
//======================================================================
void LCDInit()
{
    uint_32 i = 0;
    //从 MCU 角度,定义控制线与数据线为输出引脚,以便与 LCD 进行并行数据传输
    gpio_init(LCD_RS, 1,0);
    gpio_init(LCD_RW, 1,0);
    gpio_init(LCD_E, 1,0);
    for(i = 0;i <=7;i++)
    {
        gpio_init(LCD_D[i], 1,0);
    }
    //功能设置(命令),写指令代码
    LCDCommand(0,0x38);                        //5×7 点阵模式,两行显示,8 位数据总线
    LCDCommand(0,0x08);                        //关显示,关光标显示,不闪烁
    LCDCommand(0,0x01);                        //清屏
    for (i=0; i<80000; i++)asm("NOP");         //延时
    LCDCommand(0,0x06);
    LCDCommand(0,0x14);                        //光标右移一个字符位,AC 自动加 1
    LCDCommand(0,0x0C);                        //开显示,关光标显示,不闪烁
}

//======================================================================
//函数名称：LCDShow
//函数返回：无
//参数说明：需要显示的数据
//功能概要：通过液晶显示 data 中的 32 字节数据
//======================================================================
void LCDShow(uint_8 data[32])
{
    uint_8 i;
```

```
    //LCD 初始化
    LCDInit();

    //显示第 1 行 16 个字符,后 7 位为 DD RAM 地址(0x00)
    LCDCommand(0,0x80);
    //写 16 个数据到 DD RAM
    for (i = 0;i < 16;i++)          //将要显示在第 1 行上的 16 个数据逐个写入 DD RAM 中
    {
        LCDCommand(1,data[i]);
    }
    //显示第 2 行 16 个字符,后 7 位为 DD RAM 地址(0x40)
    LCDCommand(0,0xC0);
    for (i = 16;i < 32;i++)         //将要显示在第 2 行上的 16 个数据逐个写入 DD RAM 中
    {
        LCDCommand(1,data[i]);
    }
}

//-------------------------------以下为内部函数存放处------------------------------------
//===========================================================
//函数名称: LCDCommand
//函数返回: 无
//参数说明: cmdordata,待写命令或数据
//功能概要: 向 HD44780 写 cmdordata,且延时
//===========================================================
void LCDCommand(uint_8 RS,uint_8 cmdordata)
{
    uint_8 i,temp;
    uint_16 j;
    //发出 RS、RW 信号
    gpio_set(LCD_RS, RS);
    gpio_set(LCD_RW, 0);
    //等待延迟防止重复调用此函数而 LCD 卡死
    for (j=0; j<3600; j++);asm("NOP");
    //数据送到 LCD 的数据线上
    for(i = 0;i <=7;i++)
    {
        gpio_set(LCD_D[i], 0);
    }
    for(i = 0;i <=7;i++)
    {
        temp = 0x01 & (cmdordata>>(i));
        gpio_set(LCD_D[i], temp);
    }
    //给出 E 信号的下降沿(先高后低),使数据写入 LCD
    gpio_set(LCD_E, 1);
    for (j=0;j<25;j++) asm("NOP");
    gpio_set(LCD_E, 0);
}
//----------------------------------------内部函数结束----------------------------------
```

8.4 键盘、LED 及 LCD 驱动构件测试实例

本测试调用了键盘、LED 及 LCD 构件，在 LCD 上两行显示"The keyboard you just input is ."，采用 SysTick 定时器中断，LED 显示 8692，键盘按键的定义值将在 LCD 的右下角显示出来。网上教学资源中，还给出了键盘中断编程实例。

1. 主函数

```
//说明见工程文件夹下的 Doc 文件夹内 Readme.txt 文件
//==========================================================

#include "includes.h"                        //包含总头文件

int main(void)
{
    //1.声明主函数使用的局部变量
    uint_8 i;
    uint_8 g_temp[32]="The keyboard you just input is .";
    //2.关总中断
    DISABLE_INTERRUPTS;                      //关总中断
    //3. 初始化外设模块
    LEDInit();                               //LED 初始化
    LCDInit();                               //LCD 初始化
    KBInit();                                //键盘初始化
    systick_init(CORE_CLOCK_DIV_16, 10);     //初始化 SysTick 周期、开中断、启动计数
    //4.变量赋初值
    g_LEDBuffer[0]=8;                        //LED 缓冲区赋值
    g_LEDBuffer[1]=6;
    g_LEDBuffer[2]=9;
    g_LEDBuffer[3]=2;
    for(i=0;i<32;i++)  {g_LCDBuffer[i]=g_temp[i];}  //LCD 缓冲区赋值
    LCDShow(g_LCDBuffer);                    //LCD 显示初始字符
    //5. 使能模块中断

    //6. 开总中断
    ENABLE_INTERRUPTS;                       //开总中断
    //======================================================
    while(1)
    {
    }
    //======================================================
    return 0;
}
```

2. 中断函数服务例程

```
//==========================================================
//文件名称: isr.c
```

```
//功能概要：中断函数服务例程文件
//版权所有：苏州大学 NXP 嵌入式中心(sumcu.suda.edu.cn)
//更新记录：2012-11-12   V1.0；2016-05-08 V6.0 (WYH)
//==================================================================
#include "includes.h"

//==================中断函数服务例程==================
//==================================================================
//函数名称：SysTick_Handler(SysTick 中断函数服务例程)
//中断条件：根据 main 函数中的对 SysTick 初始化，每 10ms 产生一次 SysTick 中断，执行本程序
//功能概要：LED 刷新、扫描键盘，如有实际按键，在 LCD 的右下角显示键定义值
//==================================================================
void SysTick_Handler(void)
{
    uint_8 kb;

    LEDshow(g_LEDBuffer);                       //LED 显示刷新

    kb=KBScanN(10);                             //扫描键盘
    if (kb != 0xff)                             //有实际按键
    {
        g_LCDBuffer[31] = KBDef(kb);            //填写 LCD 缓冲区
        LCDShow(g_LCDBuffer);                   //LCD 显示
    }
}
```

小　　结

本章阐述了利用GPIO构件制作应用构件的基本方法，其目标是制作的应用构件具有可移植性与可复用性。要做到源程序代码完全可复用性，必须坚持在应用构件的源程序代码中只能出现应用对象的引线名称，在头文件中对它们进行宏定义。头文件的可移植性，是指应用对象硬件接线因使用MCU的不同引脚，在头文件中进行宏定义。

(1) 阐述了未编码键盘识别的基本问题：主要有键的识别、抖动问题与重键问题。主要给出矩阵键盘的硬件连接方式、键值含义、扫描法获取键值的基本原理、键值计算方法。要求重点掌握根据行扫描法识别按键基本原理扫描一次键盘获得键值函数KBScan1及键值转为定义值函数KBDef的编程方法。

(2) 阐述了LED数码管的基本工作原理。给出了4连排的共阴极动态数码管的连接方式和编程原理。分析了利用人眼的视觉暂留效应，动态刷新所有的显示位，实现不同的位显示内容的原理。需要注意的是，若仅用一组数据总线控制较多的显示位，由于刷新周期过长，可能出现闪烁或显示亮度较低等现象，此时若不使用多组数据总线控制，就需要考虑使用锁存器等技术改善显示效果。给出了LED数码显示的驱动构件led.h和led.c，在构件中包含模块初始化(LEDInit)、显示数组内容(LEDShow)等基本操作函数。另外，在构件中

还添加了显示码转换函数(LEDChangeCode),实现对LED数码管对可显示符号的编码。

(3) 给出了点阵字符型液晶显示模块的驱动构件设计实例。重点掌握时序,完成MCU如何向LCD中的HD44780控制器送出一个字节的命令或数据的函数LCDCommand。以便在同类应用构件设计中达到举一反三的目的。为了使程序的可复用性面提高,程序中使用指令延时的地方,适当加大,这样使得即使用于总线时钟频率较高的MCU,延时时间也能达到要求。

(4) 给出了键盘、LED及LCD构件的测试实例。网上教学资源中,还给出了键盘中断编程实例。

习　题

1. 简述行扫描法识别按键的基本原理。

2. 给出6×5键盘的键值计算方法及扫描一次键盘获得键值的函数KBScan1()设计。

3. 简述扫描法动态显示LED的基本原理,给出8个数码管的LED构件设计,说明设计动态LED构件使用静态变量的优点。

4. 根据HD44780的写操作时序,给出void LCDCommand(uint_8 cmd)的设计流程图。

5. 简要说明键盘、LED、LCD构件封装的基本要点。

6. 综合设计:参照本章综合实例,在此基础上编程。在LCD第一行显示"时:分:秒",第二行显示按键的定义值与键值。LED上显示秒。

第 9 章　Flash 在线编程

本章导读：本章阐述 MCU 内部 Flash 存储器的在线编程方法。9.1 节阐述 Flash 编程知识要素，随后给出 Flash 驱动构件头文件及使用方法。仅从应用角度，已经可以进行 Flash 在线编程的实际应用了。9.2 节介绍如何保护程序区或重要参数区，避免误擦除，还介绍了如何进行程序加密及如何去除密码。9.3 节介绍 Flash 驱动构件的设计方法，包括 Flash 模块编程结构、Flash 构件设计技术要点、Flash 构件封装要素分析及 Flash 构件源代码。

本章参考资料：本章主要参考 KL25 参考手册第 27 章 FTFA 及 26 章 FMC 部分相关内容。

9.1　Flash 驱动构件及使用方法

9.1.1　Flash 在线编程的基本概念

Flash 存储器具有固有不易失性、电可擦除、可在线编程、存储密度高、功耗低和成本较低等特点。随着 Flash 技术的逐步成熟，Flash 存储器已经成为 MCU 的重要组成部分。

Flash 存储器固有不易失性这一特点与磁存储器相似，不需要后备电源来保持数据。Flash 存储器可在线编程可以取代电可擦除可编程只读存储器(Electrically Erasable Programmable Read-Only Memory，EEPROM)，用于保存运行过程中的参数。

从 Flash 存储器的基本特点可以看出，在 MCU 中，可以利用 Flash 存储器固化程序，这一般通过编程器来完成。Flash 存储器工作于这种情况，叫作监控模式或写入器编程模式。即**通过编程器将程序写入 Flash 存储器中的模式被称为写入器编程模式**。另一方面，由于 Flash 存储器具有电可擦除功能，因此，在程序运行过程中，有可能对 Flash 存储区的数据或程序进行更新，Flash 存储器工作于这种情况，叫作用户模式或在线编程模式。即**通过运行 Flash 内部程序对 Flash 其他区域进行擦除与写入，称为 Flash 在线编程模式**。

对 Flash 存储器的读写不同于对一般的 RAM 读写，需要专门的编程过程。Flash 编程的基本操作有两种：擦除(Erase)和写入(Program)。**擦除操作的含义是将存储单元的内容由二进制的 0 变成 1，而写入操作的含义是将存储单元的某些位由二进制的 1 变成 0**。Flash 在线编程的写入操作是以字为单位进行的。在执行写入操作之前，要确保写入区在上一次擦除之后没有被写入过，即写入区是空白的(各存储单元的内容均为 0xFF)。所以，在写入之前一般都要先执行擦除操作。Flash 在线编程的擦除操作包括整体擦除和以 m 个字为单位的擦除。这 m 个字在不同厂商或不同系列的 MCU 中，其称呼不同，有的称为“块”，有的称为“页”，有的称为“扇区”等。**它表示在线擦除的最小度量单位**。

目前的 MCU，在 Flash 擦除/写入过程中一般需要高于电源的电压，称为编程电压 Vp。

有的芯片需要外部接入编程电压 Vp，有的芯片通过内部机制解决，取决于生产厂商的技术成熟度。有的 MCU 在擦除/写入加 Vp 的过程中，会出现不稳定，需要特殊处理。特殊处理的方法，一般是把待执行的机器代码设法放到 RAM 中执行，增加了编程复杂度。随着技术的发展，一些新的 MCU 芯片已经有了新的解决方案。在 KL 系列 MCU 中，在启动 Flash 命令前，只需要在杂项模块中把平台控制器(MCM_PLACR)的 ESFC 位置 1 即可。

9.1.2　KL25/26 芯片 Flash 构件头文件及使用方法

KL25/26 芯片内部 Flash 模块被简称为 FTFA。FTFA 全称是 Freescale TFS Flash A，其中，TFS 是指薄膜存储器(Thin Film Storage)，它是一种 Flash 的制作工艺，A 是指存储器容量大小。

KL 系列 MCU 芯片的 Flash 模块以扇区为基本组织单位，每个扇区的大小为 1KB。在线编程时，擦除以扇区为单位进行；内建擦除与写入算法，简化了编程过程；具有保护机制以防止意外擦除或写入。在 3.3 节中已经给出 KL25/26 芯片 128KB 的 Flash 地址范围是 0x0000_0000～0x0001_FFFF，分为 128 个扇区，实际编程时以扇区为逻辑单位。

头文件 flash.h 中给出了 Flash 驱动构件提供的 6 个基本对外接口函数，包括初始化函数、擦除、写入、读取(按逻辑地址)、读取(按物理地址)及保护操作。以写入操作为例，将写入操作这一过程中涉及的必要信息都设置为入口参数，包括扇区号、扇区内偏移地址、写入字节数目、源数据的缓冲区首地址，在使用时无须考虑写入的底层驱动实现方法。读取操作则结合实际应用场景，封装为按照逻辑地址读取(扇区号及偏移地址)和按照物理地址直接读取两种，前者方便地读取指定扇区指定偏移处的内容，后者可以方便用户直接读取指定物理地址的数据，如变量、配置域内容等，实际上，读操作可以对 RAM、Flash 及外设映像地址区进行。另外，还提供了一个判别一个区域是否为空(0xFF)的函数 flash_isempty。

1. Flash 驱动构件头文件

```
//============================================================
//文件名称：flash.h
//功能概要：flash 底层驱动构件头文件
//版权所有：苏州大学 NXP 嵌入式中心(sumcu.suda.edu.cn)
//更新记录：2013-06-06 V1.0；2016-06-06 V6.0
//适用芯片：KL25、KL26、KW01
//============================================================

#ifndef _FLASH_H
#define _FLASH_H

#include "common.h"

//============================================================
//函数名称：flash_init
//函数返回：无
//参数说明：无
//功能概要：flash 初始化
//============================================================
```

```
void flash_init();

//======================================================================
//函数名称：flash_erase_sector
//函数返回：函数执行执行状态：0=正常；1=异常。
//参数说明：sect：目标扇区号(范围取决于实际芯片,例如KL25：0~127,每扇区1KB)
//功能概要：擦除Flash存储器的sect扇区(每扇区1KB)
//======================================================================
uint_8 flash_erase(uint_16 sect);

//======================================================================
//函数名称：flash_write
//函数返回：函数执行状态：0=正常；1=异常。
//参数说明：sect：扇区号(范围取决于实际芯片,例如KL25：0~127,每扇区1KB)
//          offset:写入扇区内部偏移地址(0~1020,要求为0,4,8,12,…)
//          N：写入字节数目(4~1024,要求为4,8,12,…)
//          buf：源数据缓冲区首地址
//功能概要：将buf开始的N字节写入到Flash存储器的sect扇区的offset处
//======================================================================
uint_8 flash_write(uint_16 sect,uint_16 offset,uint_16 N,uint_8 *buf);

//======================================================================
//函数名称：flash_read_logic
//函数返回：无
//参数说明：dest：读出数据存放处(传地址,目的是带出所读数据,RAM区)
//       sect：扇区号(范围取决于实际芯片,例如KL25：0~127,每扇区1KB)
//       offset:扇区内部偏移地址(0~1020,要求为0,4,8,12,…)
//       N：读字节数目(1~1024)
//功能概要：读取Flash存储器的sect扇区的offset处开始的N字节到RAM区dest处
//======================================================================
void flash_read_logic(uint_8 *dest,uint_16 sect,uint_16 offset,uint_16 N);

//======================================================================
//函数名称：flash_read_physical
//函数返回：无
//参数说明：dest：读出数据存放处(传地址,目的是带出所读数据,RAM区)
//       addr：目标地址,要求为4的倍数(例如：0x00000004)
//       N：读字节数目(1~1024)
//功能概要：读取Flash指定地址的内容
//======================================================================
void flash_read_physical(uint_8 *dest,uint_32 addr,uint_16 N);

//======================================================================
//函数名称：flash_protect
//函数返回：无
//参数说明：regionNO：待保护区域号,实际保护扇区号为regionNO×4~regionNO×4+3
//       如保护区域号12,实际保护48,49,50,51这4个扇区
//功能概要：Flash保护操作
//说明：每调用本函数一次,保护4个扇区(regionNO×4~regionNO×4+3)
//======================================================================
void flash_protect(uint_8 regionNO);
```

```
//==========================================================
//函数名称: flash_isempty
//函数返回: 1=目标区域为空; 0=目标区域非空
//参数说明: 所要探测的 flash 区域初始地址及范围
//功能概要: Flash 判空操作
//==========================================================
uint_8 flash_isempty(uint_8 * buff, uint_16 N);

(有关内部使用的宏定义,略)
//==========================================================
#endif //_FLASH_H
```

2. Flash 驱动构件的使用方法

Flash 头文件中给出了 Flash 中 6 个最主要的基本构件函数,包括初始化函数(flash_init())、擦除(flash_erase(uint_16 sect))、写入(flash_write(uint_16 sect, uint_16 offset, uint_16 N, uint_8 * buf))、按逻辑地址读取(flash_read_logic(uint_8 * dest, uint_16 sect, uint_16 offset, uint_16 N))、按物理地址读取(flash_read_physical(uint_8 * dest, uint_32 addr, uint_16 N))、保护(flash_protect(uint_8 regionNO))。

初始化函数直接调用即可,无入口参数及返回值。擦除和写入操作类似,都返回擦除/写入的结果(正常/异常),由于擦除操作对象是整个扇区,因此入口参数仅需一个扇区号。写入操作入口参数较多,除扇区号外,还有扇区内偏移地址、写入字节数目、写入数据的缓冲区首地址。读取除了要传入读取字节数、读取后存放的目的地址以外,还需要根据是按照逻辑地址还是物理地址读取,传入相应的扇区号、偏移地址,或者直接物理地址数。

下面以向 50 扇区 0 字节开始的地址写入 30 个字节"Welcome to Soochow University!"为例。

(1) 首先,要进行初始化 Flash 模块。

```
flash_init();
```

(2) 因为执行写入操作之前,要确保写入区在上一次擦除之后没有被写入过,即写入区是空白的(各存储单元的内容均为 0xFF)。所以,在写入之前要根据情况是否先执行擦除操作,即擦除 50 扇区。

```
flash_erase_sector(50);
```

(3) 通过封装好的入口参数进行传参,进行写入操作。向 50 扇区 0 字节开始的 30 个字节内写入"Welcome to Soochow University!"。

```
flash_write(50,0,30,"Welcome to Soochow University!");
```

(4) 按照逻辑地址读取时,定义足够长度的数组变量 params,并传入数组的首地址作为目的地址参数,传入扇区号、偏移地址作为源地址,传入读取的字节长度。例如,从 50 扇区 0 字节开始的地址读取 30 字节长度字符串。

```
flash_read_logic(params,50,0,30);
```

(5) 按照物理地址直接读取时,定义足够长度的数组变量 paramsVar,并传入数组的首地址作为目的地址参数,传入直接地址数作为源地址,传入读取的字节长度。例如,从0x1ffff0a0 地址处读取存放在此处的1字节长度的全局变量值:

```
flash_read_physical(paramsVar, (uint_8 *)0x1FFFF0A0,1);
```

(6) Flash 保护函数的使用非常简单,入口参数为待保护扇区区域号(0～31)。即 KL25 的128个扇区平均分成的32个待保护区域(每个区域4个扇区,保护区域为最小的保护单位,详见9.3节)。若需要保护50扇区,仅需要调用函数 flash_protect(12),函数会将50扇区在内的4个扇区(48、49、50、51扇区)保护起来。函数无返回值。

Flash 驱动构件测试工程见网上教学资源"..\02-Software\KL25-program\ch09-Flash"文件夹。

9.2　Flash 保护与加密

9.2.1　Flash 保护含义及保护函数的使用说明

为了防止某些 Flash 存储区域受意外擦除、写入的影响,可以通过设置 FPROT 寄存器来保护这些 Flash 存储区域。保护后,该区域将无法进行擦除、写入操作。芯片复位后 Flash 区域的保护状态解除,即可恢复擦除写入的功能。

对于保护功能而言,Flash 存储器被平均分成32个可独立保护区域(区域号0～31),每个区域包含4个扇区。(详细介绍请参阅9.3.1节。)KL25 的 Flash 保护功能主要是通过对 Flash 保护寄存器 FPROTn 进行设置达到对某个特定区域的保护效果。FPROTn 一组4个8位寄存器。每个寄存器位对应于程序 Flash 存储器的平均分成的32个区域中的一个,各个区域可被单独地保护起来,以防止意外情况下被写入或擦除。

下面是 Flash 构件中 Flash 保护函数的使用说明。

```
//======================================================================
//函数名称: flash_protect
//函数返回: 无
//参数说明: regionNO: 待保护区域号,实际保护扇区号为 regionNO×4～regionNO×4+3
//            如保护区域号12,实际保护48,49,50,51这4个扇区
//功能概要: Flash 保护操作
//说明: 每调用本函数一次,保护4个扇区(regionNO×4～regionNO×4+3)
//======================================================================
void flash_protect(uint_8 regionNO)
```

9.2.2　Flash 加密方法与去除密码方法

1. Flash 加密方法

除了保护之外，为了保护开发者的权益，Kinetis 系列 MCU 还提供了加密的功能。未加密芯片，表示可以通过调试接口（如 JTAG、SWD 和 USBDM 等）访问芯片内部的存储器。加密芯片，通过外部调试接口只能进行整体擦除操作，而无法执行读取或写入 Flash 的指令。运行 MCU 内部程序对 Flash 访问则不受任何影响。

这里给出一种简单的加密方法，就是直接对启动文件 startup_MKL25Z4.S 中配置域字段进行改写，将配置域相应位置位为安全模式/不安全模式。这种方式不对芯片最终用户开放加密解密入口，对于开发者来说，操作简单直接，且更为安全。

通过对 Flash 安全配置域设置，可以配置程序 Flash 保护寄存器 FPROT、加密寄存器 FSEC 以及选项寄存器 FOPT，从而限制了对 Flash 存储空间的访问。Flash 存储器的 0x0000_0400～0x0000_0410 的地址范围共 16 个字节为 Flash 配置域。Flash 配置域的含义如表 9-1 所示。

表 9-1　Flash 配置

地址偏移量	寄存器名	大小/B	内容说明
0～7	KEY byte0～7	8	后门访问密钥
8～11	FPROT0～3	4	程序 Flash 保护寄存器
12	FSEC	1	Flash 加密寄存器
13	FOPT	1	Flash 选项寄存器
14、15	—	2	保留

在这个区段中保存了默认的 Flash 保护设定和加密属性，用以对 Flash 模块进行访问控制。在芯片复位时，芯片内部机制会自动将 Flash 配置域中相应内容对程序 Flash 保护寄存器 FPROT 和 Flash 安全寄存器 FSEC 进行设置。FSEC 寄存器详细介绍请参阅 9.3.1 节。

下面给出一个安全状态的配置域配制方法（后门密钥为 12345678）及未加密状态的配置域（默认的配置域），供参考。

```
/* Flash 配置 */
.section .FlashConfig, "a"
    /* Flash 加密的配置域，加密后无法通过外部接口访问存储器，整体擦除操作除外 */
    /*
    .long 0x04030201
    .long 0x08070605
    .long 0xFFFFFFFF
    .long 0xFFFFFF80
    */
    /* 不加密的配置域（默认） */
    .long 0xFFFFFFFF
    .long 0xFFFFFFFF
```

```
.long 0xFFFFFFFF
.long 0xFFFFFFFE
.text
.thumb
```

将上段代码中配置域设置为加密模式后，重新编译工程并下载到芯片中，复位后，芯片处于安全模式(加密状态)，MCU内部程序对Flash存储器的访问不受影响，在线擦除、写入正常。但是，若有人试图通过USBDM写入调试器对芯片进行访问操作，则会报错，提示"Device appears secured"，如图9-1所示。此时，只能进行整体擦除。

2. Flash去除密码的方法

考虑到保护开发者权益的应用场景，不推荐使用加密解密函数进行状态切换。在这种情况下，若想恢复通过外部调试接口重新对芯片进行访问，只有进行整体擦除。擦除后即可恢复空白片的状态，重新写入程序。此时，若想设置芯片的安全或不安全状态，只需要参照9.4.1节，设置相应的配置域字段即可。

USBDM烧写工具ARM Programmer提供这种整体擦除后重新烧写的功能，如图9-2所示，选择待烧写文件、芯片类型后，在设备操作一栏选择EraseMass整体擦除，即可当作空白片重新烧写程序。

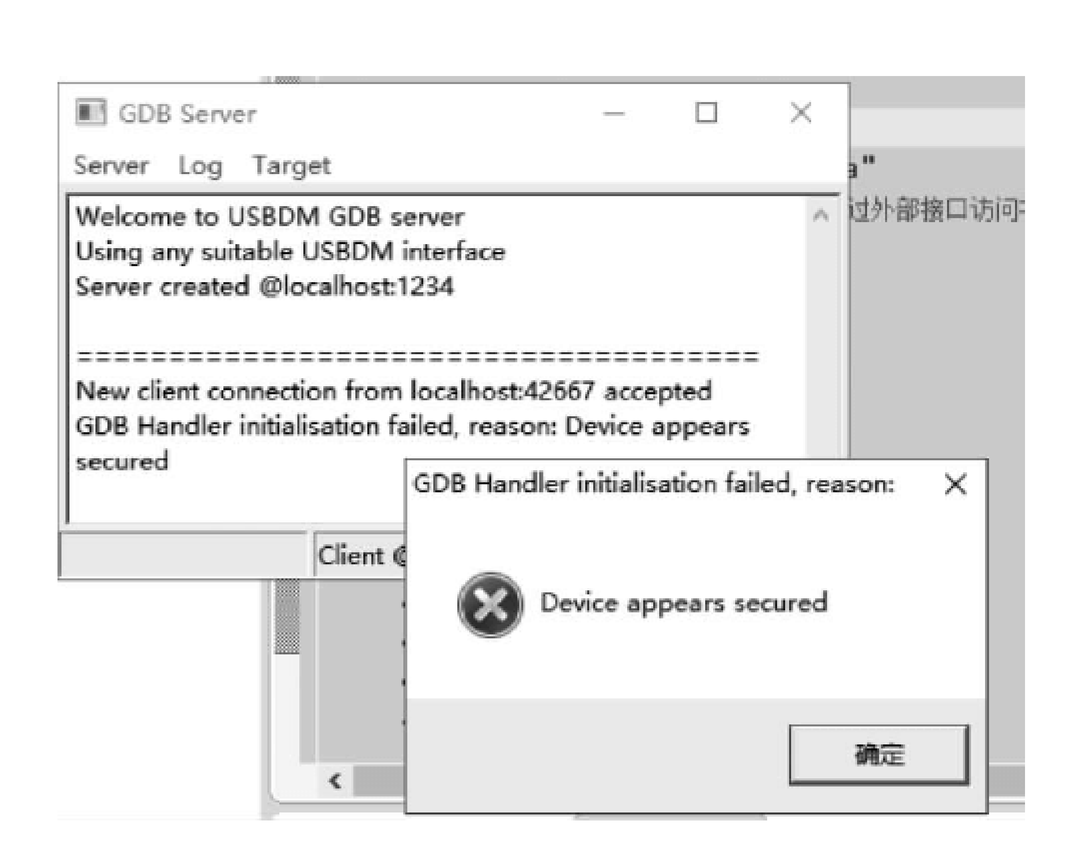


图9-1　安全状态(加密后)下外部接口访问受限

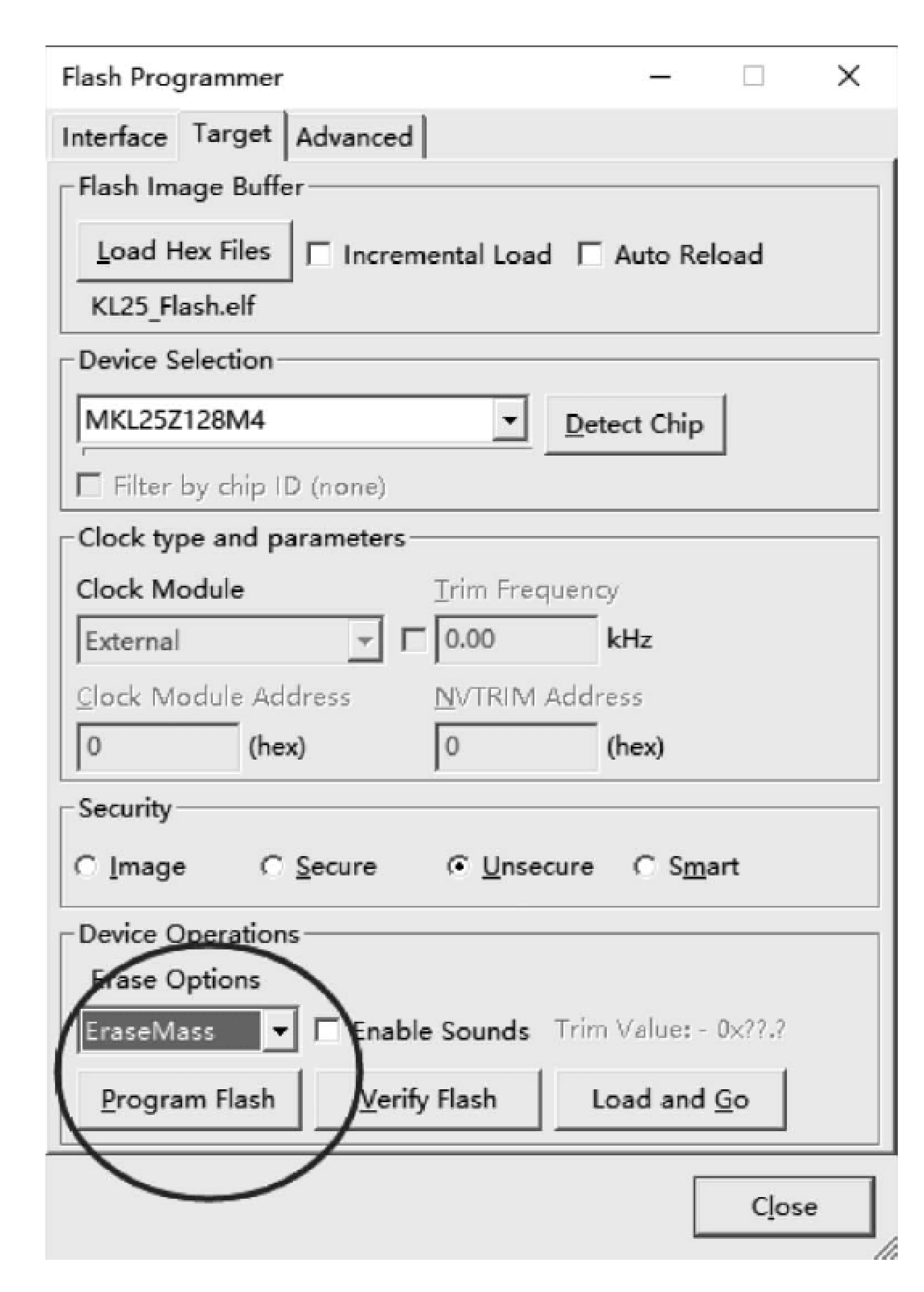


图9-2　USBDM的整体擦除后烧写功能

9.3　Flash 驱动构件的设计方法

本节讨论的是 Flash 驱动构件是如何制作出来的。首先从芯片手册中获得 Flash 模块编程结构，即用于制作 Flash 驱动构件的有关寄存器；随后从芯片手册的 Flash 模块的功能描述部分，总结为 Flash 驱动构件设计技术要点；接下来分析 Flash 驱动构件的封装要点，即根据 Flash 在线编程的应用需求及知识要素，分析 Flash 驱动构件应该包含哪些函数及哪些参数；最后给出 Flash 驱动构件的源程序代码。

9.3.1　Flash 模块编程结构

提供 Flash 在线编程的寄存器有：一个状态寄存器(FTFA_FSTAT)、一个配置寄存器(FTFA_FCNFG)、一个安全寄存器(FTFA_FSEC)、一个选项寄存器(FTFA_FOPT)、12 个通用命令参数寄存器(FTFA_FCCOBn)、4 个保护寄存器(FTFA_FPROTn)。

1. Flash 状态寄存器

Flash 状态寄存器(FTFA_FSTAT)给出 FTFA 模块的操作状态，如表 9-2 所示。CCIF、RDCOLERR、ACCERR 和 FPVIOL 这些数据位是可读可写的。MGSTATO 数据位是只读的，无符号数据位是 0，并且是不可写的。复位值为 0。

表 9-2　FIFA_FSTAT 结构

数据位	D7	D6	D5	D4	D3～D1	D0
读	CCIF	RDCOLERR	ACCERR	FPVIOL	0	MGSTAT0
写	w1c	w1c	w1c	w1c	—	

D7(CCIF)——命令完成中断标志位。CCIF 标志表示一个 Flash 命令完成。通过写 1 清除该标志位以启动一个新的命令，在命令执行期间该标志位保持为 0，命令执行完毕或终止时由存储控制器将其置 1。CCIF＝1，Flash 命令完成；CCIF＝0，当前有命令在执行。

D6(RDCOLERR)——Flash 读冲突错误标志。RDCOLERR 标志位表示 MCU 试图读取 Flash 命令正在操作的数据区域(CCIF＝0)。任何同时的访问都将被 Flash 仲裁逻辑单元定义为冲突错误。在此情况下读取的数据不能保证正确性。写 1 清除该寄存器，由存储控制器自动置该位。RDCOLERR＝1，检测到冲突错误；RDCOLERR＝0，未检测到冲突错误。

D5(ACCERR)——Flash 访问错误标志。ACCERR 标志位表示非法访问 Flash 资源或启动了非法命令。当 ACCERR 为 1 时，不能清除 CCIF 标志，以启动另一条命令。写 1 清该标志位。ACCERR＝1，检测到访问错误；ACCERR＝0，未检测到访问错误。

D4(FPVIOL)——Flash 保护区非法访问标志。FPVIOL 标志位表示试图擦除或写入保护的 Flash 区域。当 FPVIOL 为 1 时，不能清除 CCIR 标志启动命令。写 1 清该标志位。FPVIOL＝1，检测到保护区非法访问错误；FPVIOL＝0，未检测到保护区非法访问错误。

D3～D1 保留。该位段保留，只读为 0。

D0(MGSTAT0)——执行 Flash 命令出错标志。在执行命令时，可能会产生运行时错误，例如，无法验证擦除等。当产生运行时错误时，将 FSTAT[MGSTAT0]位置位。

2. Flash 配置寄存器

该寄存器提供关于当前 FTFA 模块的功能状态的信息。擦除控制数据位(ERSAREQ 和 ERSSUSP)有写入限制。

D7(CCIE)——指令完成中断使能(读/写)，CCIE 数据位在 FTFA 指令完成时控制中断的产生。当 CCIE=0，指令完成中断禁用；当 CCIE=1，指令完成中断使能。无论何时，当 FSTAT[CCIF]被设置时，产生一个中断请求。

D6(RDCOLLIE)——读冲突错误中断使能(读/写)，RDCOLLIE 数据位在 FTFA 读冲突发生时控制中断的产生。当 RDCOLLIE=0，读冲突错误中断禁用；当 RDCOLLIE=1，读冲突错误中断使能。无论何时，当 FTFA 读冲突错误被侦测到时，产生中断请求。

D5(ERSAREQ)——擦除所有块请求(只读)，这个数据位向内存控制器发出一个执行擦除所有块的请求，解除芯片的安全状态。ERSAREQ 不是直接可写的，但是可以通过用户间接控制。ERSAREQ 数据位在当一个外部的对 FTFA 和 CCIF 的擦除所有请求被激发时被设定。ERSAREQ 由 FTFA 在操作完成时清除。

D4(ERSSUSP)——擦除挂起使能位(读/写)，ERSSUSP 数据位允许用户在执行擦除 Flash 扇区命令时，挂起该操作。当 ERSSUSP=0，禁止挂起请求；当 ERSSUSP=1，允许当前 Flash 扇区擦除命令挂起。

D3～D0(保留)，只读为 0。

3. Flash 安全寄存器

该只读寄存器保留了所有跟 MCU 和 FTFA 模块相关的位。在复位的过程中，位于程序 Flash 配置域中相关的数据被装到这个寄存器中。地址：4002_0000h 基地址+2h 偏移量=4002_0002h。

D7、D6(KEYEN)——后门安全密钥使能位(只读)，该位段确定启用还是不启用访问 FTFA 模块的密钥。当 KEYEN=00，后门密钥访问禁用。当 KEYEN=01，后门密钥访问禁用(推荐使用该 KEYEN 状态来禁用后门密钥访问)；当 KEYEN=10，后门密钥访问启用；当 KEYEN=11，后门密钥访问禁用。

D5、D4(MEEN)——整体擦除使能(只读)，启用或者禁用 FTFA 模块的整体擦除功能。MEEN 的状态只有在 SEC 数据位被设为保护且 FTFA 模块在 NVM 普通模式下时才有效。当 SEC 数据位被设置为未保护时，MEEN 的设置无效。当 MEEN=00，整体擦除启用；当 MEEN=01，整体擦除启用；当 MEEN=10，整体擦除禁用；当 MEEN=11，整体擦除启用。

D3、D2(FSLACC)——恩智浦失败分析访问码(只读)，这个位段确定在恩智浦设备中返回部分失败的分析报告时，启用还是禁用访问 Flash。当 SEC 被设为安全同时 FSLACC 被设为拒绝，则对程序 Flash 内容的访问都会被拒绝。同时恩智浦工厂在测试的错误分析之前，必须执行擦除操作来解锁芯片。当访问被允许(SEC 设为不安全或 SEC 安全而 FSLACC 被设为允许)，恩智浦工厂测试就能够看到当前的 Flash 状态。FSLACC 位段的状态只有在 SEC 位段设为安全时才有关联。当 SEC 段位设为非安全，则 FSLACC 设定值

无效。当 FSLACC＝00 或 10 时，恩智浦工厂访问允许；当 FSLACC＝01 或 11 时，恩智浦工厂访问拒绝。

D1、D0(SEC)——Flash 安全位(只读)，这个位段定义了 MCU 的安全状态。在安全状态中，MCU 有限制地访问 FTFA 模块资源。这些限制依设备而不同，并且在芯片配置中加以详尽描述。如果 FTFA 模块启用后门安全密钥解密，则 SEC 位段被强制设为 10b。当 SEC＝00，MCU 处于安全状态；当 SEC＝01，MCU 处于安全状态；当 SEC＝10，MCU 处于不安全状态(标准出厂设定为 FTFA 不安全)；当 SEC＝11，MCU 处于安全状态。

4. Flash 选项寄存器

MCU 通过检测 Flash 选项寄存器的状态来制定自己的操作。Flash 选项寄存器的值在复位过程中从 Flash 配置域中载入。寄存器中的所有位均为只读。

5. Flash 通用命令参数寄存器组

FCCOB(Flash Common Command Object Registers)寄存器组存放 12 字节的命令码及相关参数。编号为 0～B(十六进制)标识的各寄存器均可进行独立配置。配置命令时，对 FCCOB 寄存器的写入顺序不做要求，但要确保赋值有效。参数的有效范围与命令有关。地址：4002_0000h 基地址＋4h 偏移量＋(1d×i)，i＝0～11。

在执行命令时，首先需配置好 FCCOB 中的字段，包括命令码及对应的参数。然后通过写 1 清除 FSTAT[CCIF]位，启动命令。此时，所有的 FCCOB 参数字段将被锁定，在命令执行期间不允许被用户修改，直到命令执行结束(CCIF＝1)。在此，不提供命令缓冲或队列的机制，只有当前命令结束之后才能载入下一个命令。

有些命令通过 FCCOB 寄存器组返回执行结果，当 FSTAT[CCIF]标志位为 1 时，FCCOB 寄存器组的返回值有效。

下面列出了 FTFA 命令的一般格式。FCCOB 寄存器组中的第一个寄存器 FCCOB0 中总是存放命令码，此 8 位的命令码指定了将要执行的命令。FCCOB1～FCCOB3 存放 Flash 地址[23:16]、[15:8]、[7:0]，FCCOB 4～FCCOB B 存放数据字节 0～7。

注意：FCCOB 寄存器组使用"大端"地址转换。对于所有大于 1 字节的命令参数，高位字节存放在较小编号的寄存器中。FCCOB 寄存器组中的寄存器可以单字节、单字(2 字节)或长字(4 字节)方式进行赋值。

6. Flash 保护寄存器

FPROT 寄存器(FTFA_FPROTn)定义了哪一块逻辑 Flash 区域不能被写入或擦除。逻辑保护的 Flash 区域不能改变自身的内容；这意味着这些区域不能被写入，也不能被任何 FTFA 指令擦除。没有被保护区域的内容可以被写入或擦除。

4 个 FPROT 寄存器允许 32 个被保护区域，保护区域是最小的保护单位。每个位保护一块 1/32 的 Flash。如果这个区域小于 32KB，那么每块被保护大小设置为 1KB。如果 Flash 存储小于等于 24KB，那么 FPROT0 不可用；如果 Flash 存储小于等于 16KB，那么 FPROT1 不可用；如果 Flash 存储小于等于 8KB，那么 FPROT2 不可用；在复位的过程中，FPROT 寄存器将 Flash 配置域中的程序 Flash 保护数据载入。寄存器中每个位段的定义如表 9-3 所示。

表 9-3　Flash 保护寄存器各位段

程序 Flash 保护寄存器	程序 Flash 保护位	Flash 配置域偏移地址
FPROT0	PROT[31:24]	0x0008
FPROT1	PROT[23:16]	0x0009
FPROT2	PROT[15:8]	0x000A
FPROT3	PROT[7:0]	0x000B

要修改在复位阶段载入的程序 Flash 保护设置，先对在 Flash 配置域中的保护程序 Flash 区块解除保护，然后更新对程序 Flash 保护位段。地址：4002_0000h 基地址+10h 偏移量+(1×i)，i=0～3。程序 Flash 保护寄存器(FTFA_FPROTn)定义如下。

D7～D0——PROT 程序 Flash 区域保护，每个程序 Flash 区域能够通过设置相关的 PROT 位以避免程序和擦除操作对其进行修改。在 NVM 正常模式：保护只能增加，就是说当前未被保护的区域可以增加为被保护区，但当前被保护区域不能解除保护。因为未被保护的区域被标记为 1，而被保护区域被标记为 0，只有当将 1 修改为 0 时是允许的。这个 1 到 0 的转换校验是按位一次进行的。FPROT 位段中 1 到 0 的转换均可被接受，而 0 到 1 的转换则被忽略。在 NVM 特别模式中，所有 FPROT 的位段都是可写的，没有任何限制未被保护的区域和被保护的区域可相互转化。限制：当一条指令在执行时(CCIF=0)，用户不得向任何 FPROT 寄存器写入数据。尝试在程序 Flash 的受保护区域中修改数据会导致保护冲突错误，并同时会置位 FSTAT[FPVIOL]。如果一个程序 Flash 块中存在保护区域，则不能对整个这个块进行操作。32 位保护寄存器中的每一位代表所有程序 Flash 的 1/32。如果存储器总大小小于 32KB 的话，那每一个保护块设定为 1KB。当 PROT=0，程序 Flash 区域被保护。当 PROT=1，程序 Flash 区域未被保护。

7. 杂项控制模块的平台控制寄存器

F000_300C 杂项控制模块的平台控制寄存器 MCM_PLACR(Platform Control Register)，地址 F000_300C。该寄存器的挂起 Flash 控制器使能位(Enable Stalling Flash Controller，ESFC)用于 Flash 擦除/写入时是否可以允许中断。在 Flash 初始化时，设置此位，可以解决早期 Flash 擦除/写入程序需移入 RAM 中运行的编程方式。

9.3.2　Flash 驱动构件设计技术要点

在 KL25 微控制器中，对 Flash 存储器的擦除操作可以进行整体擦除，也可仅擦除从某一起始地址开始的一个扇区(1KB)。也就是说，擦除的最小单位为扇区，不能仅擦除某个字节或一次擦除小于 1KB 的空间。对 Flash 存储器进行写入时，必须将一组数据准备好，擦除 Flash 存储器中相应区域后再进行写入。考虑到对 Flash 存储器的某一字节擦除/写入会影响其所在的整个扇区，所以，在进行擦除/写入操作之前，要了解当前执行程序在 Flash 中的存储位置，不要擦除运行程序所在的扇区。

1. FTFA 命令操作过程

要改变 Flash 存储器内容一般都需要通过 FTFA 命令操作实现。写 FTFA 命令的流程图如图 9-1 所示。Flash 模块对命令寄存器组(FCCOB)中的内容执行一系列的检查，只有通过检查验证无误后，命令才能被执行。启动一个命令前，FSTAT 寄存器的 ACCERR

和 FPVIOL 位必须为 0，并且 CCIF 标志必须读为 1 以确保之前的命令已经完成。当 CCIF 为 0 时，表示上一条命令仍在执行，新的命令过程无法启动，写 FCCOB 寄存器组的操作也将失效。

1）载入命令到 FCCOB 寄存器组

执行 FTFA 命令时，用户必须将需要的所有指定命令及参数存放到 FCCOB 寄存器组中。但对 FCCOB 寄存器组中寄存器的写入顺序不做要求。

2）清 CCIF 位启动命令

一旦所有相关的命令参数被载入到 FCCOB 寄存器组中，用户写 1 到 FSTAT[CCIF] 位清零，即可启动命令。CCIF 标志将保持为 0，直到 FTFA 命令执行完成。FSTAT 寄存器包含一个锁定机制，可以避免在前一命令产生访问错误（FSTAT[ACCERR]＝1）或违反保护（FSTAT [FPVIOL]＝1）时启动新的命令。在产生错误的情况下，需要两次写入 FSTAT 才能启动下一个命令：第一次写入将清错误标志位，第二次写入清 CCIF 位（启动命令）。

3）执行命令与报错

命令的处理过程包含如图 9-3 所示的步骤。

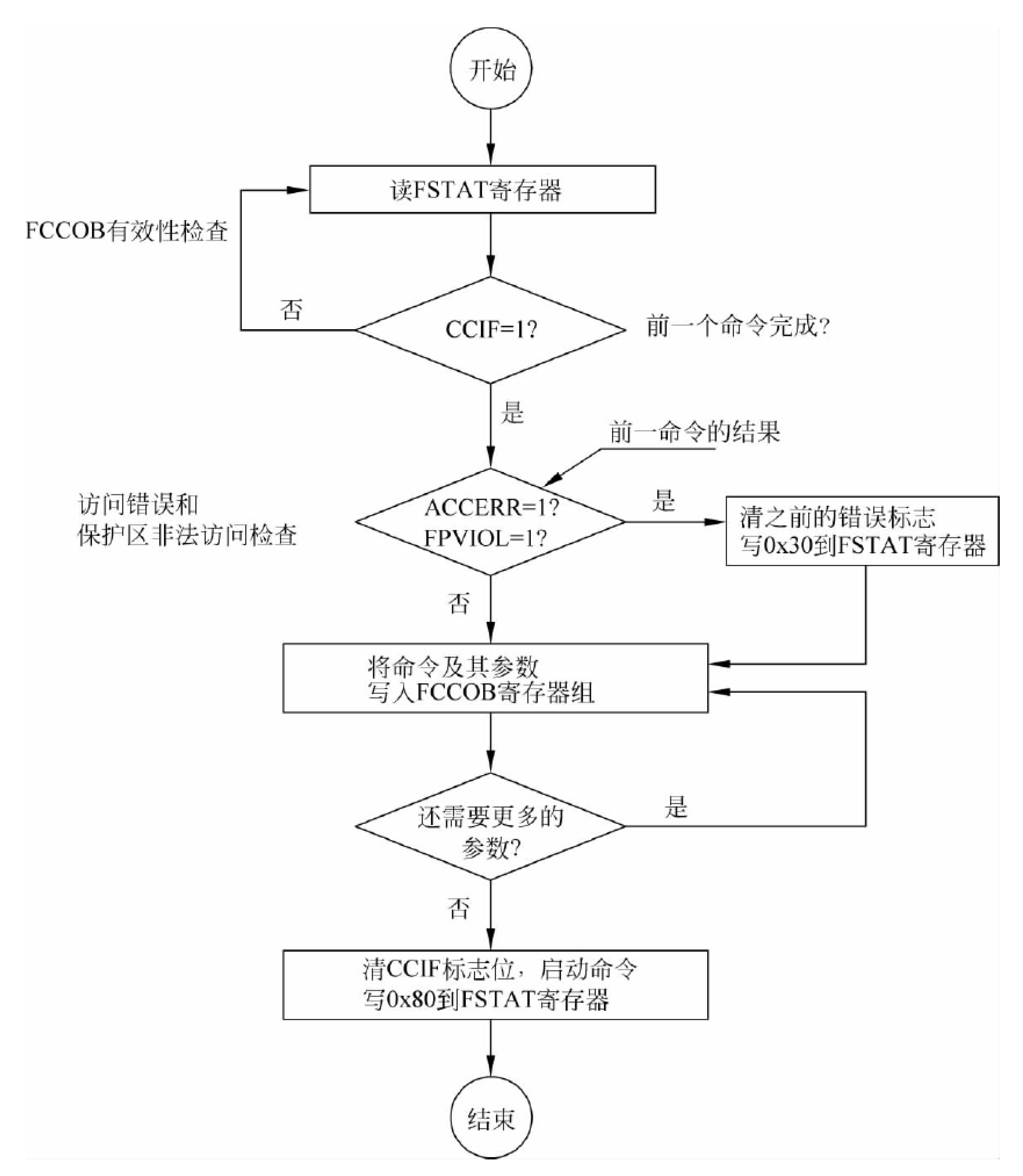


图 9-3　FTFA 命令写操作执行流程图

(1) FTFA 读取命令码并执行一系列与命令相关的参数检查和保护检查。若未通过参数检查,则将访问错误标志位 FSTAT[ACCERR]置位。ACCERR 位标识无效的指令码和超出边界地址的访问。通常情况下,访问错误是由 FCCOB 寄存器组中的参数设定无效产生的。写入和擦除命令均需要对地址进行检查,以确定操作是否在保护地址范围内执行。若未通过保护检查,则将保护错误标志位 FSTAT[FPVIOL]置位。若未通过参数设定或保护检查,命令处理过程将不能继续,此时,置 FSTAT[CCIF]位表示命令执行过程的结束。

(2) 若通过参数检查和保护检查,命令即可被执行。在执行命令时,可能会产生运行时错误,例如,无法验证擦除等。当产生运行时错误时,将 FSTAT[MGSTAT0]位置位。执行命令时,可能产生访问错误、保护错误和运行时错误,但若存在访问错误和保护错误(此时命令未被执行),将不会出现运行时错误。

(3) 命令运行的结果通过 FCCOB 和 FSTAT 寄存器反馈给用户。

(4) FTFA 自动将 FSTAT[CCIF]置位,标志命令完成。

表 9-4 中列出了所有 FTFA 命令的功能。

表 9-4 FTFA 命令功能

FCMD	命　令	功　能
0x01	读一个扇区	验证给定的程序 Flash 或数据 Flash 已经擦除
0x02	写入检查	测试上次写入的位置是否可读
0x03	读信息	从程序或数据 Flash 信息寄存器(IFR)中读 4 字节,或版本 ID
0x06	写入长字	将 4 字节写入程序 Flash 块或者数据 Flash 块中
0x09	擦除 Flash 扇区	将程序 Flash 或者数据 Flash 某个扇区全部擦除
0x40	读整个块	确保所有的程序块或数据 Flash 块已经擦除之后解锁 MCU
0x41	只读一次	在程序 Flash 块 0 IFR 专用的 64 字节中读出 4 字节数据
0x43	只写入一次	在程序 Flash 块 0 IFR 专用的 64 字节中一次编写 4 字节的程序
0x44	擦除所有块	擦除所有的程序、数据 Flash 块。之后验证擦除并且解锁 MCU。注意:只有当所有存储区不受保护时才能擦除
0x45	验证后门访问秘钥	在通过密钥比较之后再解锁 MCU

2. 扇区擦除命令

擦除 Flash 扇区操作将擦除 Flash 扇区中所有的内容。FCCOB0 存放擦除命令 0x09(ERSSCR),FCCOB1～FCCOB3 存放 Flash 擦除地址[23:16],[15:8],[7:0]。

需要注意的是,程序 Flash 的第一个扇区存放了中断向量表和 Flash 配置字段,因此,不建议在线擦除程序 Flash 第一个扇区,也不建议擦除程序操作码存放的区域。在擦除构件封装时,允许擦除 0 扇区及配置域,但配置域中设置了芯片加密相关内容,单纯擦除该扇区会使配置域中加密字段也写 FF 变为安全模式,导致芯片加密(详见 9.3.1 节),因此,为了保证擦除操作后芯片可用,需要重新对相应字段置位,使配置域恢复不安全模式(出厂设置)。

在使用 Flash 在线编程功能时,应查看链接文件确定程序在程序 Flash 中的存储位置,找出未使用的 Flash 空间作为在线 Flash 编程的可用资源才可对其编程。

例如,在本工程下的\Debug\.lst 文件中有如下说明:

Idx	Name	Size	VMA	LMA	File off	Algn
2	.text	00003d68	00000800	00000800	00008800	2 ** 3

说明 Flash 中程序存放区域开始地址为 0x00000800，程序占用区域大小为 00003d68，因此，目前程序占用了 0～17 扇区，进行 Flash 在线编程时，应避免对这前 18 个扇区进行操作。

3. 长字写入命令

长字写入命令将使用内建的算法将数据写入到程序或数据 Flash 中 4 字节区域。需要注意的是，在写入数据前，要保证写入的位置已经被擦除过。Flash 写入长字命令的 FCCOB 命令及参数形式：FCCOB0 存放 0x06，FCCOB1～FCCOB3 存放写入的 Flash 地址 [23:16]，[15:8]，[7:0]，FCCOB4～FCCOB7 存放 Byte 0～ Byte 3 的数据值。

当清 CCIF 标志启动长字写入命令时，FTFA 试图将数据内容写到 Flash 中的指定地址，并且检查保护状态位。目标 Flash 区位必须为未保护状态，从而允许执行写入命令。写入操作是单向的。只能将 Flash 中的位从擦除的状态“1”转为写入状态“0”。以 MGSTAT0 标志位标识写入“0”状态位的错误。长字写入命令执行完成后将 CCIF 标志置位。设定的起始地址必须是长字对齐(Flash 地址[1:0]＝00)。

(1) 写入数据 Byte 0 被写到起始地址 start。

(2) 写入数据 Byte 1 被写入到地址 start＋0b01。

(3) 写入数据 Byte 2 被写入到地址 start＋0b10。

(4) 写入数据 Byte 3 被写入到地址 start＋0b11。

9.3.3　Flash 驱动构件封装要点分析

Flash 具有初始化、擦除和写入、读取(按逻辑地址)、读取(按物理地址)、保护 6 种基本操作。按照构件的思想，可将它们封装成 6 个独立的功能函数，初始化函数完成对 Flash 模块的工作属性的设定，Flash 擦除、写入、读取及保护函数则完成实际的任务。对 Flash 模块进行编程，实际上已经涉及对硬件底层寄存器的直接操作，因此，可将初始化、擦除和写入等 6 种基本操作所对应的功能函数共同放置在命名为 flash.c 的文件中，并按照相对严格的构件设计原则对其进行封装，同时配以命名为 flash.h 的头文件，用来定义模块的基本信息和对外接口。

下面以 Flash 的初始化，擦除、写入、读取(按逻辑地址)、读取(按物理地址)、保护 6 种基本操作为例，来说明实现构件化编程的全过程。

1. 初始化函数 void flash_init(void)

在操作 Flash 模块前，需要对模块进行初始化，主要是判断并等待 Flash 操作命令完成、清相关标志位、杂项模块中平台控制寄存器的 PLACR_ESFC 的设置。

2. 擦除函数 uint_8 flash_erase(uint_16 sect)

由于在写入之前 Flash 字节或者长字节必须处于擦除状态(不允许累积写入，否则可能会得到意想不到的值)，因此，在写入操作前，一般先进行 Flash 的擦除操作。

擦除操作有擦除块(擦除 Flash 中所有的地址)和擦除扇区两种操作模式，在本书中，主要以擦除扇区为例。首先需要确定要擦除的扇区号，作为参数传入，最终返回擦除状态(正

常/异常)。函数过程中主要利用通用命令参数寄存器组来控制擦除的命令、地址范围及擦除数据(写0xFF)。

3. 写入函数 uint_8 flash_write(uint_16 sect,uint_16 offset,uint_16 N,uint_8 * buf)

写入函数与擦除函数类似,主要区别在于,擦除操作向目标地址中写0xFF,而写入操作需要写入指定数据。因此,写入操作的入口参数较多,包括目标扇区号、写入扇区内部偏移地址、写入字节数目以及源数据缓冲区首地址。写入后返回写入状态(正常/异常)。

4. 读取(按逻辑地址)void flash_read_logic(uint_8 * dest,uint_16 sect,uint_16 offset,uint_16 N)

按照逻辑地址读取的操作需要将Flash中指定扇区、指定偏移量的指定长度数据读取、存放到另一个地址中,方便上层函数调用,因此,函数需要包括一个目的地址变量作为入口参数,此外,还包括扇区号、偏移字节数、读取长度。

5. 读取(按物理地址)void flash_read_physical(uint_8 * dest,uint_32 addr,uint_16 N)

按照物理地址直接读取和按照逻辑地址读取类似,需要一个目的地址作为复制的目标地址,需要给定读取长度作为入口参数,但是直接读取只需要一个源地址即可,省去了扇区号与偏移地址的计算过程,更为简单,也便于读取存放在RAM区的全局变量等内容。

6. 保护函数 void flash_protect(uint_8 regionNO);

保护函数因芯片特性,只能允许对整个对齐区域(4个扇区)的保护,但是对于拥有32个区域的KL25Z128来说,已经允许了很大的保护灵活性。因此设计保护函数时,需要确定待保护的区域号作为入口参数。实际保护时,程序会对该区域号在内的4个对齐扇区进行保护。

9.3.4 Flash 驱动构件的源程序代码

Flash驱动构件源程序文件flash.c文件内容如下。

```
//包含头文件
#include "flash.h"
#include "string.h"//调用函数memcpy需包含此头文件

//==================内部调用函数声明==================
static uint_32 flash_cmd_launch(void);
//==========================================================

//====================外部接口函数====================
//==========================================================
//函数名称: flash_init
//函数返回: 无
//参数说明: 无
//功能概要: 初始化flash模块
//==========================================================
void flash_init(void)
{
    //等待命令完成
    while(!(FTFA_FSTAT & CCIF));
```

```
    //清除访问出错标志位
    FTFA_FSTAT = ACCERR | FPVIOL;

    //置杂项模块中平台控制寄存器的 PLACR_ESFC,Flash 模块擦写保护
    //BSET(MCM_PLACR_ESFC_SHIFT,MCM_PLACR);
    /*
    //清杂项模块中平台控制寄存器的 PLACR_ESFC,Flash 模块擦写不保护
    BCLR(MCM_PLACR_ESFC_SHIFT,MCM_PLACR);          //(实验观察现象用)
    */
}
//===========================================================================
//函数名称:flash_erase_sector
//函数返回:函数执行执行状态:0=正常;1=异常。
//参数说明:sect:目标扇区号(范围取决于实际芯片,例如 KL25:0~127,每扇区 1KB)
//功能概要:擦除 flash 存储器的 sect 扇区(每扇区 1KB)
//===========================================================================
uint_8 flash_erase(uint_16 sect)
{
    union
    {
        uint_32  word;
        uint_8   byte[4];
    } dest;

    dest.word = (uint_32)(sect * (1<<10));

    //设置擦除命令
    FTFA_FCCOB0 = ERSSCR;                          //擦除扇区命令

    //设置目标地址
    FTFA_FCCOB1 = dest.byte[2];
    FTFA_FCCOB2 = dest.byte[1];
    FTFA_FCCOB3 = dest.byte[0];

    //执行命令序列
    if(1 == flash_cmd_launch())                    //若执行命令出现错误
        return 1;                                  //擦除命令错误

    //Flash 存储器起始地址为 0x0000_0400 到 0x0000 _0410 的 16 个字节,为 Flash 配置域
    //若擦除地址小于 0x400 时,会擦除配置域内容为全 F,导致芯片加密(安全模式),
    //因此需要重新写入配置域,将 FSEC 寄存器 SEC 安全位重新置位为不安全模式(解密)
    if(dest.word < 0x400)
    {
        //写入 4 字节
        FTFA_FCCOB0 = PGM4;
        //设置目标地址
        FTFA_FCCOB1 = 0x00;
        FTFA_FCCOB2 = 0x04;
        FTFA_FCCOB3 = 0x0C;
        //数据
```

```
        FTFA_FCCOB4 = 0xFF;
        FTFA_FCCOB5 = 0xFF;
        FTFA_FCCOB6 = 0xFF;
        FTFA_FCCOB7 = 0xFE;             //FTFA_FSEC 寄存器 SEC 位为 10(不安全模式)
        //执行命令序列
        if(1 == flash_cmd_launch())     //若执行命令出现错误
            return 2;                   //解密错误
    }

    return 0;                           //成功返回
}
//==========================================================================
//函数名称: flash_write
//函数返回: 函数执行状态: 0=正常; 1=异常。
//参数说明: sect: 扇区号(范围取决于实际芯片,例如 KL25: 0~127,每扇区 1KB)
//          offset:写入扇区内部偏移地址(0~1020,要求为 0,4,8,12,…)
//          N: 写入字节数目(4~1024,要求为 4,8,12,…)
//          buf: 源数据缓冲区首地址
//功能概要: 将buf 开始的 N 字节写入到 flash 存储器的 sect 扇区的 offset 处
//==========================================================================
uint_8 flash_write(uint_16 sect,uint_16 offset,uint_16 N,uint_8 *buf)
{
    uint_32 size;
    uint_32 destaddr;

    union
    {
        uint_32   word;
        uint_8   byte[4];
    } dest;

    if(offset%4 != 0)
        return 1;                         //参数设定错误,偏移量未对齐(4 字节对齐)

    //设置写入命令
    FTFA_FCCOB0 = PGM4;
    destaddr = (uint_32)(sect * (1<<10) + offset);   //计算地址
    dest.word = destaddr;
    for(size=0; size<N; size+=4, dest.word+=4, buf+=4)
    {
        //设置目标地址
        FTFA_FCCOB1 = dest.byte[2];
        FTFA_FCCOB2 = dest.byte[1];
        FTFA_FCCOB3 = dest.byte[0];

        //复制数据
        FTFA_FCCOB4 = buf[3];
        FTFA_FCCOB5 = buf[2];
        FTFA_FCCOB6 = buf[1];
        FTFA_FCCOB7 = buf[0];
```

```
        if(1 == flash_cmd_launch())
            return 2;                       //写入命令错误
    }

    return 0;                               //成功执行
}

//==========================================================================
//函数名称：flash_read_logic
//函数返回：无
//参数说明：dest：读出数据存放处(传地址，目的是带出所读数据，RAM 区)
//          sect：扇区号(范围取决于实际芯片，例如 KL25：0~127，每扇区 1KB)
//          offset:扇区内部偏移地址(0~1020，要求为 0,4,8,12,…)
//          N：读字节数目(4~1024，要求为 4,8,12,…)
//功能概要：读取 flash 存储器的 sect 扇区的 offset 处开始的 N 字节，到 RAM 区 dest 处
//==========================================================================
void flash_read_logic(uint_8 *dest,uint_16 sect,uint_16 offset,uint_16 N)
{
    uint_8 *src;
    src=(uint_8 *)(sect*1024+offset);      //计算地址
    memcpy(dest,src,N);                     //从 src 复制 N 字节数据到 dest
}

//==========================================================================
//函数名称：flash_read_physical
//函数返回：无
//参数说明：dest：读出数据存放处(传地址，目的是带出所读数据，RAM 区)
//          addr：目标地址，要求为 4 的倍数(例如：0x00000004)
//          N：读字节数目(0~1023)
//功能概要：读取 flash 指定地址的内容
//==========================================================================
void flash_read_physical(uint_8 *dest,uint_32 addr,uint_16 N)
{   uint_8 *src;
    src=(uint_8 *)addr;
    memcpy(dest,src,N);                     //从 src 复制 N 字节数据到 dest
}

//==========================================================================
//函数名称：flash_protect
//函数返回：无
//参数说明：regionNO：待保护区域号，实际保护扇区号为 regionNO×4~regionNO×4+3
//       如保护区域号 12，实际保护 48,49,50,51 共 4 个扇区
//功能概要：flash 保护操作
//说明：每调用本函数一次，保护 4 个扇区(regionNO×4~regionNO×4+3)
//==========================================================================
void flash_protect(uint_8 regionNO)
{
    uint_8 offset;
```

```
    uint_8 regionCounter;
    offset=regionNO%8;                    //获得偏移,即保护位号
    regionCounter=regionNO/8;             //获得应置位的寄存器号
    //FTFA_FPROT 寄存器组: 4002_0010h+(1×i),i=0~3,0 对应 FTFA_FPROT3
    BCLR(offset, *((uint_8 *)(0x40020010+regionCounter)));
}

//======================================================================
//函数名称: flash_isempty
//函数返回: 1=目标区域为空; 0=目标区域非空。
//参数说明: 所要探测的 flash 区域初始地址
//功能概要: flash 判空操作
//======================================================================
uint_8 flash_isempty(uint_8 *buff,uint_16 N)
{
   uint_16 i,flag;
   i=0;
   flag=1;
   for(i = 0; i<N; i++)                   //遍历区域内字节
   {
      if(buff[i]!=0xff)                   //非空
      {
          flag=0;
          break;
       }
   }
   return flag;
}

//--------------------------------以下为内部函数存放处--------------------------------------
//======================================================================
//函数名称: flash_cmd_launch
//函数返回: 0-成功 1-失败
//参数说明: 无
//功能概要: 启动 Flash 命令
//======================================================================
static uint_32 flash_cmd_launch(void)
{
    //清除访问错误标志位和非法访问标志位
    FTFA_FSTAT = ACCERR | FPVIOL;

    //启动命令
    FTFA_FSTAT = CCIF;

    //等待命令结束
    while(!(FTFA_FSTAT & CCIF));

    //检查错误标志
```

```
    if(FTFA_FSTAT & (ACCERR | FPVIOL | MGSTAT0))
        return 1 ;                          //执行命令出错
    return 0; //执行命令成功
}
//===================================================================
```

小　　结

本章介绍了 Flash 存储器的在线编程方法、保护配置方法及 Flash 内程序加密及去除密码方法。

(1) Flash 存储器具有掉电后数据不丢失这一重要特点。通过编程器将程序写入 Flash 存储器中的模式被称为写入器编程模式。通过运行 Flash 内部程序对 Flash 其他区域进行擦除与写入，称为 Flash 在线编程模式。Flash 存储器具有在线编程功能，可以使用 Flash 取代电可擦除可编程只读存储器 EEPROM，用来保存运行过程中期望掉电后不会丢失的参数。Flash 编程的基本操作有两种：擦除和写入。擦除操作的含义是将存储单元的内容由二进制的 0 变成 1，而写入操作的含义是将存储单元的某些位由二进制的 1 变成 0。Flash 在线编程的写入操作是以字为单位进行的。在执行写入操作之前，要确保写入区在上一次擦除之后没有被写入过，即写入区是空白的。

KL25 芯片 Flash 模块以扇区为基本组织单位，每个扇区的大小为 1KB。KL25 芯片内部 Flash 的起始地址是 0x0000_0000，有 128 个扇区。Flash 在线编程中的擦除操作是以扇区为逻辑单位。

Flash 在线编程的驱动构件封装了 6 个基本对外接口函数，包括初始化函数、擦除、写入、读取(按逻辑地址)、读取(按物理地址)及保护操作函数。

(2) Flash 保护是为了防止某些 Flash 存储区域受意外擦除、写入的影响。保护后，该区域将无法进行擦除、写入操作。芯片复位后 Flash 区域的保护状态解除，即可恢复擦除写入的功能。对于保护功能而言，Flash 存储器被平均分成 32 个可独立保护区域(区域号 0～31)，每个区域包含 4 个扇区。关于芯片加密问题，对未加密芯片，表示可以通过调试接口(如 JTAG、SWD 和 USBDM 等)访问芯片内部的存储器。对于加密芯片，通过外部调试接口只能进行整体擦除操作，而无法执行读取或写入 Flash 的指令。运行 MCU 内部程序对 Flash 访问则不受任何影响。本章给出了一种简单的加密方法，就是直接对启动文件 startup_MKL25Z4.S 中配置域字段进行改写，将配置域相应位置位为安全模式/不安全模式。

(3) 在 Flash 驱动构件的设计方法一节，讨论了 Flash 驱动构件是如何制作出来的。首先从芯片手册中获得 Flash 模块编程结构，即用于制作 Flash 驱动构件的有关寄存器；随后从芯片手册的 Flash 模块的功能描述部分，总结为 Flash 驱动构件设计技术要点；接下来分析 Flash 驱动构件的封装要点，即根据 Flash 在线编程的应用需求及知识要素，分析 Flash 驱动构件应该包含哪些函数及哪些参数；最后给出 Flash 驱动构件的源程序代码。

习　题

1. 简要阐述 Flash 在线编程的基本含义及用途。

2. 给出 Flash 驱动构件的基本函数及接口参数。

3. 说明 Flash 保护含义，给出 Flash 驱动构件中保护函数的使用说明。

4. 说明 Flash 加密的基本含义，给出 Flash 加密及去除密码的基本方法。

5. 参考网上教学资源中的样例，编制程序，将自己的一寸照片存入 Flash 中适当区域，并重新上电复位后再读出到 PC 屏幕显示。

第10章　ADC、DAC与CMP模块

本章导读：本章主要阐述了模/数转换(ADC)、数/模转换(DAC)以及比较器(CMP)模块的工作原理和编程方法。主要内容有：10.1节在简要阐述ADC编程要素基础上，给出KL25芯片ADC知识要素及技术要点，给出ADC构件接口函数说明及使用方法举例，这样就可以对KL25芯片ADC模块进行编程操作，还给出了ADC驱动构件的制作方法；10.2节在简要阐述DA编程要素基础上，给出KL25芯片DA知识要素及技术要点，给出DA构件接口函数说明及使用方法举例，这样就可以对KL25芯片DA模块进行编程操作，还给出了DA驱动构件的制作方法；10.3节在简要阐述CMP编程要素基础上，给出KL25芯片CMP知识要素及技术要点，给出CMP构件接口函数说明及使用方法举例，这样就可以对KL25芯片CMP模块进行编程操作，还给出了CMP驱动构件的制作方法。期望通过这一章的学习，掌握嵌入式系统ADC、DAC和CMP程序的设计。

本章参考资料：10.1节AD部分寄存器参考自《KL参考手册》第28章AD模块；10.2节DA部分寄存器参考自《KL参考手册》第30章DA模块；10.3节参考自《KL参考手册》的第29章CMP模块。

10.1　模拟/数字转换器ADC

10.1.1　模/数转换器ADC的通用基础知识

AD转换模块(Analog To Digital Convert Module)即模/数转换模块，其功能是将电压信号转换为相应的数字信号。实际应用中，这个电压信号可能由温度、湿度、压力等实际物理量经过传感器和相应的变换电路转化而来。经过AD转换后，MCU就可以处理这些物理量。图10-1给出了一个数字控制系统的示意框图。

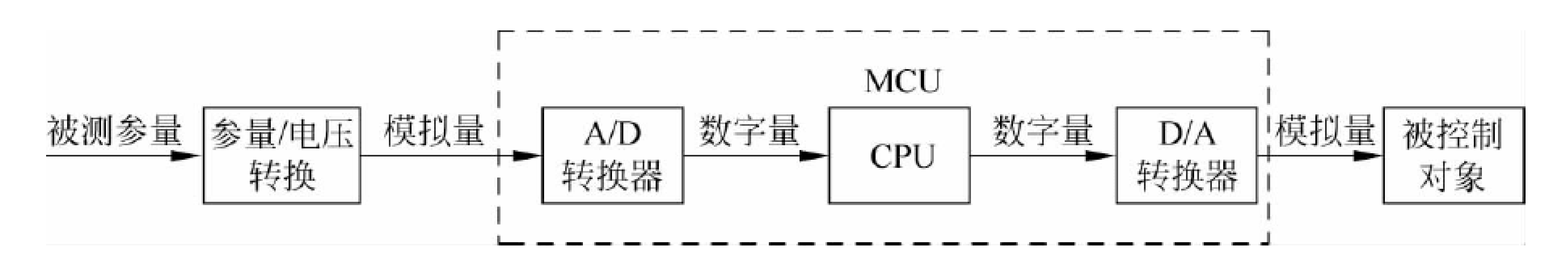


图10-1　数字控制系统组成框图

1. 与AD转换编程直接相关的基本问题

学习AD转换的编程，应该了解与AD转换编程直接相关的一些基本问题，主要有转换精度、转换速度、是单端输入还是差分输入、AD参考电压、滤波问题、物理量回归等。下面简要概述。

1) 转换精度

转换精度就是指数字量变化一个最小量时模拟信号的变化量，也称为分辨率

(Resolution),通常用AD转换器的位数来表征。AD转换模块的位数通常有8位、10位、12位、14位、16位等。设采样位数为N,则最小的能检测到的模拟量变化值为$1/2^N$。例如,某一AD转换模块是12位,若参考电压为5V(即满量程电压),则可检测到的模拟量变化最小值为$5/2^{12}=1.22$(mV),就是这个AD转换器的实际精度(分辨率)了。

2) 转换速度

转换速度通常用完成一次AD转换所要花费的时间来表征。转换速度与AD转换器的硬件类型及制造工艺等因素密切相关。其特征值为纳秒级。AD转换器的硬件类型主要有逐次逼近型、积分型、Σ-Δ调制型等。

3) 单端输入与差分输入

单端输入只有一个输入引脚,使用公共地GND作为参考电平。这种输入方式的优点是简单,缺点是容易受干扰,由于GND电位始终是0V,因此AD值也会随着干扰而变化。

差分输入比单端输入多了一个引脚,AD采样值是两个引脚的电平差值(VIN+、VIN-两个引脚电平相减),优点是降低了干扰,缺点是多用了一个引脚。通常两根差分线会布在一起,因此受到的干扰程度接近,引入AD转换引脚的共模干扰①,在进入AD内部电路时会被减掉,从而降低了干扰。

4) AD参考电压

AD转换需要一个参考电平。比如要把一个电压分成1024份,每一份的基准必须是稳定的,这个电平来自于基准电压,就是AD参考电压。粗略的情况下,AD参考电压使用给芯片功能供电的电源电压。更为精确的要求下,AD参考电压使用单独电源,要求功率小(在mW级即可),但波动小(例如0.1%),一般电源电压达不到这个精度,否则成本太高。

5) 滤波问题

为了使采样的数据更准确,必须对采样的数据进行筛选去掉误差较大的毛刺。通常采用中值滤波和均值滤波来提高采样精度。所谓中值滤波,就是将M次连续采样值按大小进行排序,取中间值作为滤波输出。而均值滤波,是把N次采样结果值相加,然后再除以采样次数N,得到的平均值就是滤波结果。若要得到更高的精度,可以通过建立其他误差模型分析方式来实现。

6) 物理量回归

在实际应用中,得到稳定的AD采样值以后,还需要把AD采样值与实际物理量对应起来,这一步称为物理量回归。AD转换的目的是把模拟信号转化为数字信号,供计算机进行处理,但必须知道AD转换后的数值所代表的实际物理量的值,这样才有实际意义。例如,利用MCU采集室内温度,AD转换后的数值是126,实际它代表多少温度呢?如果当前室内温度是25.1℃,则AD值126就代表实际温度25.1℃。

物理量回归与仪器仪表"标定"一词的基本内涵是一致的,但那里不涉及AD转换概念,只是与标准仪表进行对应,以便使得待标定的仪表准确。而物理量回归是指AD采样值如何与实际物理量值对应起来,也需借助标准仪表,从这个意义上理解,它们的基本内涵一致。设AD值为x,实际物理量为y,物理量回归需要寻找它们之间的函数关系:$y=f(x)$。

---

① 共模干扰往往是指同时加载在各个输入信号接口端的共有的信号干扰。采用屏蔽双绞线并有效接地、采用线性稳压电源或高品质的开关电源、使用差分式电路等方式可以有效地抑制共模干扰。

2. 最简单的 AD 转换采样电路

这里给出一个最简单的 AD 转换采样电路，以表征 AD 转换应用中的硬件电路的基本原理示意。下面以光敏/温度传感器为例。

光敏电阻器是利用半导体的光电效应制成的一种电阻值随入射光的强弱而改变的电阻器；入射光强，电阻减小，入射光弱，电阻增大。光敏电阻器一般用于光的测量、光的控制和光电转换(将光的变化转换为电的变化)。通常，光敏电阻器都制成薄片结构，以便吸收更多的光能。当它受到光的照射时，半导体片(光敏层)内就激发出电子-空穴对，参与导电，使电路中电流增强。一般光敏电阻器结构如图 10-2(a)中的左图所示。



图 10-2　光敏/热敏电阻器及其采样电路

与光敏电阻类似，温度传感器是利用一些金属、半导体等材料与温度有关的特性制成的，这些特性包括热膨胀、电阻、电容、磁性、热电势、热噪声、弹性及光学特征，根据制造材料将其分为热敏电阻传感器、半导体热电偶传感器、PN 结温度传感器和集成温度传感器等类型。热敏电阻传感器是一种比较简单的温度传感器，其最基本的电气特性是随着温度的变化自身阻值也随之变化。图 10-2(a)中的右图是 NTC 热敏电阻器。

在实际应用中，将光敏或热敏电阻接入图 10-2(b)的采样电路中，光敏或热敏电阻和一个特定阻值的电阻串联，由于光敏或热敏电阻会随着外界环境的变化而变化，因此 AD 采样点的电压也会随之变化，AD 采样点的电压为

$$V_{\mathrm{A/D}} = \frac{x}{R_{热敏} \times x} \times V_{\mathrm{REF}}$$

式中，x 是一特定阻值，根据实际光敏或热敏电阻的不同而加以选定。

以热敏电阻为例，假设热敏电阻阻值增大，采样点的电压就会减小，AD 值也相应减小；反之，热敏电阻阻值减小，采样点的电压就会增大，AD 值也相应增大。所以采用这种方法，MCU 就会获知外界温度的变化。如果想知道外界的具体温度值，就需要进行物理量回归操作，也就是通过 AD 采样值，根据采样电路及热敏电阻温度变化曲线，推算当前温度值。

灰度传感器也是由光敏元件构成。所谓灰度也可认为是亮度，简单地说就是色彩的深浅程度。灰度传感器的主要工作原理是它使用两只二极管，一只为发白光的高亮度发光二极管，另一只为光敏探头。通过发光管发出超强白光照射在物体上，通过物体反射回来落在光敏二极管上，由于照射在它上面的光线强弱的影响，光敏二极管的阻值在反射光线很弱(也就是物体为深色)时为几百千欧，一般光照度下为几千欧，在反射光线很强(也就是物体颜色很浅，几乎全反射时)为几十欧。这样就能检测到物体的颜色的灰度了。

本书网上教学资源中的补充阅读材料给出了一种较为复杂的电阻型传感器采样电路设计。

10.1.2 ADC驱动构件及使用方法

1. ADC的引脚与通道号

KL25的ADC模块只有一个，记为ADC0，是线性逐次逼近ADC，最高精度(分辨率)为16位。同时具有差分输入和单端输入两种采集模式。LQFP封装80引脚的MKL25Z128VLK4芯片，具有2路差分模式引脚与14路单端模式引脚。差分模式的精度可配置为16位、13位、11位、9位。单端模式的精度可配置为16位、12位、10位、8位。另外，还有其他形式的模拟输入通道，如ADC模块内包含一个温度传感器，它的输出信号接在ADC模拟量输入通道上，通道号位26。ADC输入通道情况如表10-1所示。

表10-1　MKL25Z128VLK4芯片ADC通道输入表

通道号SC1[ADCH]		差分输入DIFF=1	单端输入DIFF=0	引脚号	引脚名
十进制	二进制				
0	00000	ADC0_DP0/ADC0_DM0	ADC0_SE0	13/14	PTE20/PTE21
1,2	00001,00010	保留	保留		
3	00011	ADC0_DP3/ADC0_DM3	ADC0_SE3	15/16	PTE22/PTE23
4	00100	保留	ADC0_SE4a	14	PTE21
7	00111	保留	ADC0_SE7a	16	PTE23
4	00100	保留	ADC0_SE4b	21	PTE29
5	00101	保留	ADC0_SE5b	74	PTD1
6	00110	保留	ADC0_SE6b	78	PTD5
7	00111	保留	ADC0_SE7b	79	PTD6
8	01000	保留	ADC0_SE8	43	PTB0
9	01001	保留	ADC0_SE9	44	PTB1
10	01010	保留	保留		
11	01011	保留	ADC0_SE11	57	PTC2
12	01100	保留	ADC0_SE12	45	PTB2
13	01101	保留	ADC0_SE13	46	PTB3
14	01110	保留	ADC0_SE14	55	PTC0
15	01111	保留	ADC0_SE15	56	PTC1
10～22	10000～10110	保留	保留		
23	10111	保留	ADC_SE23	22	PTE30
24,25	11000,11001	保留	保留		
26	11010	片内温度传感器(差分)	片内温度传感器		
27	11011	Bandgap(Diff)[1]	Bandgap(S.E)①		
28	11100	保留	保留		
29	11101	VREFSH	VREFSH		
30	11110	VREFSL	保留		
31	11111	模块禁止	模块禁止		

① PMC模块带隙参考电压，详见KL25数据手册5.2.2。

2. ADC 驱动构件基本要点分析

AD 模块具有初始化、采样、滤波等操作。按照构件化的思想，可将它们封装成独立的功能函数。AD 构件包括头文件 adc. c 和 adc. h 文件。AD 构件头文件中主要包括相关宏定义、AD 的功能函数原型说明等内容。AD 构件程序文件的内容是给出 AD 各功能函数的实现过程。

在 adc. h 中，给出了用于定义 AD 采样次数的宏定义、输入模式(单端、差分输入)的宏定义和 A/B 通道组的通道选择的宏定义。注意表 10-1 中通道号相同，但单端输入对应的组不同(末尾标识 a，b)，其对应引脚不同。

除此之外，给出了两个 AD 模块必要的两个函数初始化与读取一次转换结果的函数。

1) AD 模块初始化函数 adc_init()

该函数中需要使用 4 个参数：

```
void adc_init(uint_8 chnGroup, uint_8 diff, uint_8 accurary, uint_8 HDAve);
```

chnGroup：通道组选择。在 adc. h 中定义了两个对应的宏常数供选择：MUXSEL_A(A 通道)；MUXSEL_B(B 通道)。

diff：输入模式选择。定义了两个对应的宏常数供选择：AD_DIFF(差分模式)；AD_SINGLE(单端模式)；

accurary：采样精度。差分模式下支持 9、13、11、16 这 4 种精度；单端模式下支持 8、12、10、16 这 4 种精度。

HDAve：硬件滤波次数，定义了 4 个对应的宏常数供选择：SAMPLE4、SAMPLE8、SAMPLE16、SAMPLE32，分别对应 4/8/16/32 次硬件滤波。

2) AD 模块读取一次经过硬件滤波后的值函数 adc_read()

```
uint_16 adc_read(uint_8 channel);
```

该函数仅有一个参数 channel，即所需读 AD 转换值的通道号，通道号的选择详见表 10-1，MKL25Z128VLK4 芯片 ADC 通道输入表。使用这个函数之前，需调用初始化函数对相应通道进行初始化。

3. ADC 驱动构件头文件

```
//==================================================================
//文件名称：adc.h
//功能概要：adc 底层驱动构件头文件
//更新记录：20130505,V01; 20150116,V02
//==================================================================

#ifndef _ADC_H                        //防止重复定义(开头)
#define _ADC_H

#include "common.h"                   //包含公共要素头文件
```

```
//用于定义硬件滤波次数
#define SAMPLE4    0
#define SAMPLE8    1
#define SAMPLE16   2
#define SAMPLE32   3
//定义输入模式
#define AD_DIFF         1        //差分输入
#define AD_SINGLE       0        //单端输入
//通道选择
#define MUXSEL_A        0        //选择端口的 A 通道
#define MUXSEL_B        1        //选择端口的 B 通道

//=====================================================
//函数名称:adc_init
//功能概要:初始化一个 AD 通道组
//参数说明:chnGroup:通道组;有宏常数:MUXSEL_A(A 通道);MUXSEL_B(B 通道)
//          diff:差分选择。=1,差分;=0,单端;也可使用宏常数 AD_DIFF/AD_SINGLE
//          accurary:采样精度,差分可选 9-13-11-16;单端可选 8-12-10-16
//          HDAve:硬件滤波次数,从宏定义中选择 SAMPLE4/SAMPLE8/ SAMPLE16/
//                                                    SAMPLE32
//=====================================================
void adc_init(uint_8 chnGroup,uint_8 diff,uint_8 accurary,uint_8 HDAve);

//=====================================================
//函数名称:adc_read
//功能概要:进行一个通道的一次 AD 转换
//参数说明:channel:见 MKL25Z128VLK4 芯片 ADC 通道输入表
//=====================================================
uint_16 adc_read(uint_8 channel);

#endif
```

4. ADC驱动构件使用方法

ADC驱动构件的头文件(adc.h)中包含的内容有:给出两个对外服务函数的接口说明及声明,函数包括ADC初始化函数(adc_init)、读取通道数据(adc_read)。

现在,以采集并输出KL25芯片温度为例,介绍ADC构件的使用方法。步骤如下。

(1) 初始化A通道,单端输入,16位精度,32次硬件滤波的AD转换:

```
adc_init(MUXSEL_A, AD_SINGLE,16,SAMPLE32);
```

(2) 读取通道26,每次采集32次硬件滤波,赋给16位无符号整型变量advalue:

```
advalue = adc_read(26);
```

(3) 将读取到的AD值通过公式转换成温度(公式详见KL25芯片手册28.4.8节):

```
float VTemp, temp
VTemp = (advalue * 3300) >> 16;     //或直接用(advalue × 3300) / 65536.0
temp = 25-(VTemp-719) / 1.715;
```

(4) 在串口调试工具观察温度传感器输出的温度：

```
printf("%f", temp);
```

5. ADC 驱动构件测试实例

测试工程功能概述如下。

(1) 串口通信格式：波特率 9600，1 位停止位，无校验。

(2) 上电或按复位按钮时，调试串口 1 输出“This is ADC Test!”。

(3) 主循环中，改变 RUN_LIGHT_BLUE 的小灯状态(蓝灯闪烁)。调试串口输出 AD 模块中 19 个通道的 AD 值，当配置精度为 10 位时，AD 值范围为 0～1023；当配置精度为 12 位时，AD 值范围为 0～4095；当配置精度为 16 位时，AD 值范围为 0～65 535，在这里，使用 16 位精度。选取精度取决于所需，位数低，精度低，但转换速度快。

(4) 使用串口 1 连接 PC，打开 AD 模块附带的上位机样例程序，左侧坐标系会显示芯片随时间变化的曲线图，右侧的文本框会显示各个通道采集到的十六进制数据。

ADC 构件的测试工程与上位机程序位于网上教学资源中的“..\ch10-ADC-DAC-CMP(KDS)\KL25-ADC”文件夹。

ADC 测试主函数文件 main.c 代码如下。

```
//说明见工程文件夹下的 Doc 文件夹内 Readme.txt 文件
//==========================================================

#include "includes.h"                              //包含总头文件

int main(void)
{
    //1. 声明主函数使用的变量
    uint_32   mRuncount;                           //主循环计数器
    uint_16   ADCResult[21];                       //存放 AD 结果
    uint_16   i;
    //2. 关总中断
    DISABLE_INTERRUPTS;

    //3. 初始化外设模块
    light_init(LIGHT_BLUE, LIGHT_ON);              //蓝灯初始化
    uart_init(UART_1, 9600);                       //使能串口 1,波特率为 9600
    uart_init(UART_2, 9600);                       //使能串口 2,波特率为 9600
    uart_send_string(UART_1, "This is ADC Test!\r\n"); //串口发送提示
    printf("Hello Uart2!\r\n");
    //4. 给有关变量赋初值
    mRuncount=0;                                   //主循环计数器
    //5. 使能模块中断
    uart_enable_re_int(UART_1);                    //使能串口 1 接收中断
    uart_enable_re_int(UART_2);                    //使能串口 2 接收中断
    //6. 开总中断
    ENABLE_INTERRUPTS;
```

```
    //进入主循环
    //主循环开始==================================
    for(;;)
    {
        //运行指示灯(LIGHT)闪烁--------------------------------------------------
        mRuncount++;                             //主循环次数计数器+1
        if (mRuncount >= COUNTER_MAX)            //主循环次数计数器大于设定的宏常数
        {
            mRuncount=0;                         //主循环次数计数器清零
            light_change(LIGHT_BLUE);            //蓝色运行指示灯状态变化
        }
        //以下加入用户程序----------------------------------------
        //A组初始化(通道组、单端输入,采样精度,硬件均值)
        adc_init(MUXSEL_A,0,16,SAMPLE32);
        //加头标志
        ADCResult[0] = 0x1122;
        //采集数据
        ADCResult[1] = adc_read(0);
        for (i=2;i<=8;i++)  ADCResult[i] = adc_read(i+1);
        for (i=9;i<=13;i++)  ADCResult[i] = adc_read(i+2);
        ADCResult[14] = adc_read(23);
        ADCResult[15] = adc_read(26);            //芯片温度采集通道
        //B组初始化(通道组、单端输入,采样精度,硬件均值)
        adc_init(MUXSEL_B,0,16,SAMPLE32);
        //采样
        for (i=16;i<=20;i++)  ADCResult[i] = adc_read(i-12);
        //加末尾标志
        ADCResult[20] = 0x8899;
        //将采集的A/D值通过串口发送到PC
        uart_sendN(UART_TEST,42,ADCResult);
        Delay_ms(50);
    }//主循环end_for
    //主循环结束==================================
}
```

10.1.3　ADC模块的编程结构

KL25的AD转换模块有27个寄存器,包括4个状态控制寄存器、两个配置寄存器、两个ADC数据结果寄存器、两个ADC比较值寄存器、一个ADC偏移量校正寄存器、一个ADC正向增益寄存器、一个ADC负向增益寄存器、7个ADC正向增益通用校准值寄存器、7个ADC负向增益通用校准值寄存器。下面首先介绍相关名词解释,随后介绍常用ADC寄存器。

转换完成标志:指示一个AD转换是否完成,仅当AD转换完成后才能从寄存器中读取数据。

通道:ADC模块有专门的AD转换通道,分别对应着芯片的不同引脚,读取相应引脚的数据相当于读取了通道的数据。

硬件触发:靠外部硬件的脉冲触发。

软件触发:软触发是靠软件编程的方式触发启动,一旦程序编写好了,触发启动是自动

的有规律的，除非修改程序，否则无法根据自己的意愿随意触发。

FIFO 队列：用于保存采集的 AD 数据的先进先出的队列。

KL25 的 ADC 模块的寄存器的地址在芯片头文件中。

1. ADC 状态控制寄存器

1）状态控制寄存器 ADC0_SC1A 和 ADC0_SC1B

状态控制寄存器 ADC0_SC1A 有软件触发和硬件触发两种操作模式，状态控制寄存器 ADC0_SC1B 为只用于硬件触发操作模式，它们的结构如表 10-2 所示。当 SC1A 有效控制一个转换并且处于取消当前转换时，可以对 SC1A 进行写操作。在软件触发模式下（SC2[ADTRG]=0），对寄存器 SC1A 进行写的时候会开始一个新的转换。在软件触发操作模式下不能用 SC1B 寄存器，因此对 SC1B 进行写操作不会引起一个新的转换。

表 10-2　ADC0_SC1A 和 ADC0_SC1B 结构

数据位	D31～D8	D7	D6	D5	D4～D0
读/写	0	COCO	AIEN	DIFF	ADCH
复位	0				1

D31～D8——保留位，只读，且各位值为 0。

D7——COCO，转换完成标志位，只读。当不设置比较功能（SC2[ACFE]=0）时，或不设置硬件均值功能（SC3[AVGE]=0）时，每次转换完成时置该位为 1；当比较功能使能（SC2[ACFE]=1）时，只要比较结果为真，转换完成后，该位为 1；当设置硬件均值功能（SC3[AVGE]=1）时，且均值滤波次数（该值由 SC3[AVGS]段决定）设定后，则该位为 1；当校准次序完成，则该位为 1。当对寄存器 SC1A 进行写操作或者对转换结果寄存器 Rn 进行读操作时，都会清除 COCO。

D6——AIEN，中断使能位。当 AIEN 位为 1 时，设置 COCO 位为 1 就会引发一个中断。当 AIEN 为 0 时，无动作。

D5——DIFF，差分模式使能位。当 DIFF 为 0 时，单端转换；当 DIFF 为 1 时，差分转换。在差分模式下，当 ADC 配置有效时，该模式会自动从不同通道中选择一个通道，改变转换算法和周期数完成转换。

D4～D0——ADCH，输入通道选择位。用于选择一个输入通道，见表 10-1 描述。

2）状态控制寄存器 ADC0_SC2

状态控制寄存器 ADC0_SC2 具有转换执行状态、硬件/软件触发选择、比较功能和 ADC 模块的参考电压选择等功能，其结构如表 10-3 所示。复位后，各位均为 0。

表 10-3　ADC0_SC2 结构

数据位	D31～D8	D7	D6	D5	D4	D3	D2	D1、D0
读/写	0	ADACT	ADTRG	ACFE	ACFGT	ACREN	DMAEN	REFSEL

D31～D8——保留位，只读，且各位值为 0。

D7——ADACT，转换执行位。提示一个转换或者硬件计算均值命令是否正在执行。当 ADACT=1 时，转换正在执行；当 ADACT=0 时，转换没有在执行。

D6——ADTRG,转换触发选择位。有两种触发方式,当ADTRG=1时,硬件触发。ADC硬件触发来自实时中断(RTI)计数器的输出,RTI计数器溢出触发AD转换;当ADTRG=0时,在这种模式下,写SC1(ADCH位不全为1)启动转换。

D5~D3用于转换结果与比较值寄存器CV1和CV2的比较关系,D5=1使能比较,D4=1转换结果大于等于CV1,D3=1使能范围比较,表10-4给出了比较关系。若使能比较,只有比较结果为真时,才会将转换结果存入结果寄存器,转换完成标志位COCO才会置1。

表10-4 比较模式

D5 ACFE	D4 ACFGT	D3 ACREN	CV1与CV2的关系	比较功能描述
1	0	0	CV1	转换结果<CV1,比较为真
1	1	0	CV1	转换结果≥CV1,比较为真
1	0	1	CV1≤CV2	转换结果<CV1,或者转换结果>CV2,比较为真
1	0	1	CV1>CV2	CV1>转换结果>CV2,比较为真
1	1	1	CV1≤CV2	CV1≤转换结果≤CV2,比较为真
1	1	1	CV1>CV2	转换结果≥CV1,或者转换结果≤CV2,比较为真

D2——DMAEN,DMA使能位。当DMAEN=0时,DMA禁止;当DMAEN=1时,DMA使能,同时在ADC转换完成期间会保持DMA请求。

D1、D0——REFSEL,参考电压选择位。00:选择芯片的V_{REFH}和V_{REFL}两个引脚作为AD转换的参考电压。01:可选的参考电压对。10,11:保留。

3) 状态控制寄存器ADC0_SC3

状态控制寄存器ADC0_SC3控制ADC模块的校验,持续性转换和硬件计算均值功能,其结构如表10-5所示。复位后,各位均为0。

表10-5 ADC0_SC3结构

数据位	D31~D8	D7	D6	D5、D4	D3	D2	D1、D0
读/写	0	CAL	CALF	0	ADCO	AVGE	AVGS

D31~D8——保留位,只读,且各位值为0。

D7——CAL,校验位。CAL置位后,校验开始执行,校验完成后,该位清零。必须检查CALF位来确定校验结果是否正确,因为校验一旦开始,不能被写操作中断,否则转换结果出错,导致CALF位被置位。所以CAL=1时,可以取消当前的任何转换。

D6——CALF,校对失败标志位。显示校验后的结果是否正确。当CALF=0时,校验正常;当CALF=1时,校验失败。若SC2[ADTRG]=1时,校对失败,此时任何寄存器都可以进行写操作,或者在校验过程完成之前有停止模式进入。对CALF写1,可以清除该位。

D5、D4——保留位,只读,且各位值为0。

D3——ADCO,持续转换使能位。当ADCO=0时,硬件计算均值功能使能时(AVGE=1),在开始一个转换之后接下来只有一个转换或者一组转换;当ADCO=1时,硬件计算均值功能使能时(AVGE=1),在开始一个转换之后接下来有持续的转换或多组转换。

D2——AVGE，硬件计算均值功能位。当 AVGE＝0 时，硬件计算均值功能禁止；当 AVGE＝1 时，硬件计算均值功能使能。

D1、D0——AVGS，硬件计算均值选择位。AVGS 段确定对多少个 ADC 转换结果来求平均值，进而得到 ADC 转换的平均值。00～11 分别代表 4，8，16，32 个采样均值。

2. ADC 配置寄存器

1）配置寄存器 ADC_CFG1

配置寄存器 ADC_CFG1 可以选择操作模式，设置时钟源、时钟分频，并对低功耗或者长时间采样模式进行配置，其结构如表 10-6 所示。

表 10-6　ADC_CFG1 结构

数据位	D15～D8	D7	D6、D5	D4	D3	D2	D1、D0
读/写	0	ADLPC	ADIV	ADLSMP	MODE		ADICLK
复位	0					1	

D31～D8——保留位，只读，且各位值为 0。

D7——ADLPC，低功耗配置位。ADLPC 控制连续近似值转换器的电压配置。当 ADLPC＝0 时，正常供电配置；当 ADLPC＝1 时，以最大时钟速率的代价降低功耗。

D6、D5——ADIV，时钟分频选择位。ADIV 选择 ADC 使用的分频系数产生内部时钟 ADCK。当 ADIV 分别为 00、01、10、11 时，对应的分频系数分别为 1、2、4、8，时钟频率为输入时钟、输入时钟/2、输入时钟/4、输入时钟/8。

D4——ADLSMP，采样时间配置位。ADLSMP 会根据选择的转换模式选择不同的采样次数。该位能够根据采样周期进行调整，高阻抗输入以达到精确采样或者低阻抗输入达到最大转换速率。如果持续转换使能，同时不要求高转换率，则长时间采样也可以用在更低的功耗状态下进行。当 ADLSMP＝1 时，即长时间采样选择位置位，可以选择长时间采样的范围；当 ADLSMP＝0 时，即短时间采样选择位置位，可以选择短时间采样的范围。

D3、D2——MODE，转换模式选择位。选择 ADC 采样模式。当 SC1[DIFF]＝0 时，MODE＝00、01、10、11 时，分别为单端 8 位、10 位、12 位、16 位转换；当 SC1[DIFF]＝1 时，MODE＝00、01、10、11 时，分别为带有二进制补码输出的 9 位、13 位、28 位、16 位差分转换。

D1、D0——ADICLK，输入时钟选择位。输入时钟源产生内部时钟 ADCK。当选择 ADACK 为时钟源时，不要求提前开始转换。当选择该位的同时又不需要提前开始转换（ADACKEN＝0）时，异步时钟在转换开始时有效，在转换结束时关闭。这种情况下每次时钟源再次有效时，都有一个相关的时钟开始时间延时。当 ADICLK＝00、01、10、11 时，输入时钟分别对应总线时钟、总线时钟/2、交替时钟（ALTCLK）、异步时钟（ADACK）。

2）配置寄存器 ADC_CFG2

配置寄存器 ADC_CFG2 为高速转换选择特定的高速配置，并在长采样模式下选择长时间持续采样，其结构如表 10-7 所示。复位后，各位均为 0。

表 10-7　ADC_CFG2 结构

数据位	D31～D5	D4	D3	D2	D1、D0
读/写	0	MUXSEL	ADACKEN	ADHSC	ADLSTS

D31～D5——保留位,只读,且各位值为0。

D4——MUXSEL,ADC复用选择位。当MUXSEL=0时,选择ADC_SExa通道;当MUXSEL=1时,选择ADC_SExb通道,见表10-1。

D3——ADACKEN,异步时钟输出使能位。ADACKEN可以使能异步时钟源,时钟源时钟输出和输入时钟选择的状态无关。根据MCU的配置,其他模块可以使用异步时钟。即使当ADC处于空闲或者来自不同时钟源的操作正在执行,都可设置该位允许时钟使能。同样,如果ADACK时钟已经在运行,选择带有异步时钟的简单转换或者第一个连续转换操作的延时就会减少。当ADACKEN=0时,异步时钟输出禁止;当ADACKEN=1时,不管ADC的状态是什么,异步时钟和输出时钟都有效。

D2——ADHSC,高速配置位。通过改变转换时序来允许更高速率的转换时钟(两个ADCK被加进转换时间)。当ADHSC=0时,选择正常转换时序;当ADHSC=1时,选择高速转换时序。

D1、D0——ADLSTS,长采样时间选择位。当选择了长采样时间(CFG1[ADLSMP]=1)时,ADLSTS选择扩展采样时间中的一个。该特点允许高阻抗输入,可以达到精确采样或在低阻抗输入时,可以将转换速度最大化。如果不要求高转换率,当持续转换使能时,更长的采样时间以降低功耗。其中,默认最长采样时间为4个ADCK周期。当ADLSTS=00时额外增加20个ADCK周期,01时额外增加12个ADCK周期,10时额外增加6个ADCK周期,11时额外增加两个ADCK周期,所以总共有24个、16个、10个、6个ADCK周期的采样时间。

3. ADC数据结果寄存器

KL25有两个数据结果寄存器RA,RB。其中,RA寄存器的地址为4003_B010h,RB寄存器的地址为4003_B014h。数据结果寄存器包含一个ADC转换结果,这个结果是通过通道状态控制寄存器(SC1A,SC1B)选择产生的。对于每个通道状态控制寄存器,都有一个相符合的数据结果寄存器。

在无符号右对齐模式下,结果寄存器Rn中没有使用的位会被清除,而在有符号扩展的二进制补码模式下会携带符号位(MSB)。例如,当配置成10位的单端模式时,D[15:10]会被清除。当配置成11位的差分模式时,D[15:10]会携带符号位,也就是第10位扩展成第15位。表10-8描述了数据结果寄存器在不同模式下的行为。

表10-8 数据结果寄存器描述

转换模式	各位描述	格式	
16位差分模式	D15=S,D14～D0=D	有符号的二进制补码	说明: S:符号位或者符号位扩展; D:数据(二进制补码显示)。 D31～D16保留位,只读,且各位值为0
16位的单端模式	D15～D0=D	无符号的右对齐	
13位的差分模式	D15～D12=S,D11～D0=D	扩展的有符号二进制补码	
12位的单端模式	D15～D12=0,D11～D0=D	无符号右对齐	
11位的差分模式	D15～D10=S,D9～D0=D	扩展的有符号二进制补码	
10位的单端模式	D15～D10=0,D9～D0=D	无符号右对齐	
9位的差分模式	D15～D9=S,D8～D0=D	扩展的有符号二进制补码	
8位的单端模式	D15～D9=0,D8～D0=D	无符号右对齐	

4. ADC 比较值寄存器

KL25 有两个比较值寄存器 CV1,CV2。当比较功能使能时(SC2[ACFE]=1),可以与转换结果的值做比较,见表 10-4 的比较模式。D31～D16 保留位,只读且值为 0。D15～D0 为比较值。一旦比较功能使能,只有当转换结果在和 CV1 和 CV2 比较之后结果为真的情况下才会将转换完成标志(SC1n[COCO])置位。因此,可以使用比较值寄存器来过滤一些在特定情况下一定为异常值的 AD 值,减少 CPU 被占用的时间,提高工作效率。

5. ADC 偏移量校正寄存器

在执行 ADC 转换操作之前必须校正或写入一个有效的校准值,即偏移量校正寄存器(OFS)在每次复位之后或在 AD 转换之前都需要一个给定值。该值可以是用户自定义校准偏移量或者硬件自校准偏移量(16 位左对齐二进制补码数据)。

在自校准序列完成后,OFS 将会根据校准要求自动被赋值。用户也可根据需要重写 OFS 校准值。若要初始化自校准序列,需要先将 SC2 设置为软件触发(SC2[ADTRG]=0),然后置 SC3[CAL]位,则开始硬件自校准。在自校准序列完成后,SC1n[COCO]将会被置位。在检查 SC3[CALF]位没有被置位时,则表明校准序列已经成功完成。

ADC 转换结果与该偏移量相减所得数据锁存至结果寄存器 Rn。例如,在 8 位单端模式中,最后得到的数据就是数据结果寄存器 D[7:0]减去 OFS[14:7]的值,当 OFS 中的值位为负(OFS[15]表示符号)时,则相当于将 OFS 的值与数据结果寄存器的值相加;在 16 位单端模式中,由于 OFS 没有更多的空间可以表示正负符号,必须舍去最低位,因此在这种情况下,该寄存器无法区分诸如+1 或-1 这种特殊值。若校准后的采样数据超出量程范围,其结果由当前采样模式强制输出最小/最大值。对于单端输入而言,最小输出 0x0000,对于差分而言,最小输出 0x8000。其结构如表 10-9 所示,复位默认 0x0100。

表 10-9　ADC0_OFS 结构

数据位	D31～D16	D15～D3	D2	D1、D0
读	0	OFS		
写				
复位	0		1	0

6. 用于差分模式的特殊寄存器

1) ADC 正向增益寄存器(ADC0_PG)

正向增益寄存器(PG)存放差分模式或单端模式下的正向增益校正误差,ADC0_PG 结构如表 10-10 所示。PG 实际是一个增益调整因子(16 位二进制数据格式),介于 ADPG15～ADPG14 且带小数点。用户必须根据校对步骤中描述的值对寄存器进行写操作,否则校正达不到要求。校对步骤中描述可详见参考手册 28.4.6。

表 10-10　ADC0_PG 结构

数据位	D31～D16	D15	D14～D10	D9	D8～D0
读写	0	PG			
复位	0	1	0	1	0

2）ADC 负向增益寄存器(ADC0_MG)

负向增益寄存器(MG)存放差分模式下的负向增益校正误差，在单端模式下该寄存器无效，其结构如表 10-11 所示。MG 实际是一个增益调整因子(16 位二进制数据格式)，介于 ADMG15～ADMG14 且带小数点。用户必须根据校对步骤中描述的值对寄存器进行写操作，否则校正达不到要求。校对步骤中描述可详见参考手册 28.4.6。

表 10-11　ADC0_MG 结构

数据位	D31～D16	D15	D14～D10	D9	D8～D0
读写	0	MG			
复位	0	1	0	1	0

3）ADC 正向增益通用校准值寄存器(ADC0_CLPx)

正向增益通用校准值寄存器(CLPx)存放校验功能生成的校验信息，这些寄存器带有 7 个不同宽度的校验值：CLP0[5：0]、CLP1[6：0]、CLP2[7：0]、CLP3[8：0]、CLP4[9：0]、CLPS[5：0]和 CLPD[5：0]，其结构如表 10-12 所示。一旦自校验次序确定(CAL 被清零)，CLPx 会自动置位。

表 10-12　ADC0_CLPx 结构

数据位	D31～D6	D5、D4	D3	D2	D1	D0
读写	0	CLPD				
复位	0		1	0	1	0

4）ADC 负向增益通用校准值寄存器(ADC0_CLMx)

负向增益通用校准值寄存器(CLMx)存放校验功能生成的校验信息，这些寄存器带有 7 个不同宽度的校验值：CLM0[5：0]、CLM1[6：0]、CLM2[7：0]、CLM3[8：0]、CLM4[9：0]、CLMS[5：0]和 CLMD[5：0]，其结构如表 10-13 所示。一旦自校验次序确定(CAL 被清零)，CLMx 会自动置位。

表 10-13　ADC0_CLMx 结构

数据位	D31～D6	D5、D4	D3	D2	D1	D0
读写	0	CLMD				
复位	0		1	0	1	0

10.1.4　ADC 驱动构件的设计

本节主要介绍如何根据 ADC 模块的各个寄存器的功能，结合上文给出的 adc.h 编写具体的 ADC 的驱动。

1. KL 系列 MCU 的 ADC 模块功能概述

KL25 的 ADC 模块具有单端输入与差分输入功能。当 KL25 的 ADC 配置为差分模式时，2 对差分引脚视为差分输入源，将该引脚的电压差值模数转换的测量值，而且相应的结果寄存器会出现符号位。当差分引脚 DADP 的电压比 DADM 高时，符号位为 0。当差分引脚 DADP 的电压比 DADM 低时，符号位为 1。虽然差分可以带符号，但这里还是建议读者设置 DADP 端的电压高于 DADM 端的电压值。如果 ADC 模块配置为非差分模式时，ADC0_DP0、ADC0_DP3 两个差分模式下，单端输入可视为非差分的单端输入端。其实非差分的输入可以理解为另一端电压值永远为 0V 的差分输入。

在复位、低功耗停止模式或者是当 SC1n 中的 ADCH 各位都为高时，ADC 模块是禁止的，具体请参阅电源管理信息。当一个转换完成后另外一个转换还没有开始的时候，ADC 模块处于空闲的状态。当 ADC 空闲时，异步时钟输出使能禁止，或者 CFG2[ADACKEN] 位为 0，ADC 模块处于最低功耗状态。一旦软件选定通道，ADC 模块能执行模拟信号到数字信号的转换。所有的模式根据一系列的线性逼近算法执行转换操作。

当转换完成时，转换的结果保存到数据结果寄存器中。如果中断使能(SC1n[AIEN]=1)，各自的转换完成，SC1n[COCO]位置位为 1，就会产生一个中断。

ADC 模块还具有如下功能。

(1) ADC 模块具有自动和比较寄存器(CV1 和 CV2)转换结果比较的功能。设置 SC2[ACFE]=1，在任何转换模式和配置信息下，都可以使能比较功能。一旦比较功能使能，只有当转换结果在和 CV1 和 CV2 比较之后结果为真的情况下才会将转换完成标志(SC1n[COCO])置位。因此，可以使用比较值寄存器来过滤一些在特定情况下一定为异常值的 AD 值，减少 CPU 被占用的时间，提高工作效率。

(2) ADC 模块具有将多次转换的结果求均值的功能。设置 SC3[AVGE]位，在任何转换模式和配置信息下，都可以使能硬件计算均值功能。

(3) 硬件计算均值：ADC 转换有硬件触发和软件触发两种触发方式。当硬件触发事件发生，ADC 完成规定次数的转换后，会对每次的转换结果求均值，这样做是为了防止干扰，避免错误操作。

(4) 为了满足精确定位的要求，ADC 模块还具有用芯片校验功能和 ADC 偏移量修正功能，以提高 ADC 模块的精度。可以参考 10.1.3 节中的 ADC 偏移量校正寄存器，或是参阅 KL25 参考手册的 28.4.6 节及 28.4.7 节。

实现简单的 AD 转换编程主要涉及以下几个寄存器，状态控制寄存器(ADC0_SC1、ADC0_SC3)，配置寄存器(ADC0_CFG1、ADC0_CFG2)。其中，状态控制寄存器 ADC0_SC1 用于选择转换模式(单端或者差分)、使能或禁止转换完成中断、选择转换的输入通道；状态控制寄存器 ADC0_SC3 用于选择硬件均值使能、硬件采样次数(4 次采样)；配置寄存器 ADC0_CFG1 用于选择转换模式(10 位)、总线时钟 4 分频、输入时钟选择总线时钟/2；配置寄存器 ADC0_CFG2 用于选择高速配置。基本编程步骤如下。

(1) 打开 ADC 模块时钟源，KL25 只有一个模块 ADC0，只需初始化 SIM_SCG6。

(2) 配置寄存器 ADC0_CFG1，选择精度、总线时钟 4 分频(时钟分频选择位，即 CFG1[ADIV]=10)、总线时钟/2(输入时钟选择位，即 CFG1[ADICLK]=01)。

(3) 配置状态控制寄存器 ADC0_SC3，使能硬件均值(SC3[AVGE]=1)，并以 4 次采样

求均值为准(SC3[AVGS]=00)。

(4) 更新配置寄存器ADC0_CFG2,选择高速采样(CFG2[ADHSC]=1)。

(5) 配置状态控制寄存器ADC0_SC1A,选择ADC采用通道号,通过SC1[ADCH]。

配置完上述寄存器后,ADC模块已经开始工作了,但是还需读取数据结果寄存器(ADC_R)才能得到具体的AD转换数据。

(6) 若要读取AD转换数据,通过判断状态的寄存器ADC_SC1的COCO位,判断一次转换是否完成。若完成,便读取结果寄存器ADC_R的数据位,获取具体的数据。

2. ADC驱动构件源码

```
//==========================================================
//文件名称: adc.c
//功能概要: ADC底层驱动构件源文件
//版权所有: 苏州大学恩智浦嵌入式中心(sumcu.suda.edu.cn)
//更新记录: 2013-4-7    V1.0
//==========================================================

#include "adc.h"
//内部函数声明
void adc_cal();

//==========================================================
//函数名称: adc_init
//功能概要: 初始化一个AD通道组
//参数说明: chnGroup: 通道组; 有宏常数: MUXSEL_A(A通道); MUXSEL_B(B通道)
//          diff: 差分选择。=1,差分; =0,单端; 也可使用宏常数AD_DIFF/AD_SINGLE
//          accurary: 采样精度,差分可选9-13-11-16; 单端可选8-12-10-16
//          HDAve: 硬件滤波次数,从宏定义中选择SAMPLE4/SAMPLE8/ SAMPLE16/
//                                                         SAMPLE32
//==========================================================
void adc_init(uint_8 chnGroup, uint_8 diff, uint_8 accurary, uint_8 HDAve)
{
    uint_8 ADCCfg1;
    //1.打开ADC0模块时钟
    SIM_SCGC6 |= SIM_SCGC6_ADC0_MASK;
    //2.配置CFG1寄存器:正常功耗,总线时钟4分频,总线时钟/2,常采样时间
    //2.1 根据采样精度确定ADC_CFG1_MODE位
    switch(accurary)
    {
        case 8:case 9:
            ADCCfg1 = ADC_CFG1_MODE(0);
            break;
        case 12:case 13:
            ADCCfg1 = ADC_CFG1_MODE(1);
            break;
        case 10:case 11:
            ADCCfg1 = ADC_CFG1_MODE(2);
            break;
        default:
```

```
            ADCCfg1 = ADC_CFG1_MODE(3);
            break;
        }
    //2.2 继续计算配置值(正常功耗,总线时钟 4 分频,总线时钟/2,常采样时间)
      ADCCfg1 |=  (ADC_CFG1_ADIV(2) | ADC_CFG1_ADICLK(1)
                    | ADC_CFG1_ADLSMP_MASK);
    //2.3 进行配置
    ADC0_CFG1 = ADCCfg1;
    //3.根据通道组,配置 CFG2 寄存器
    //3.1 配置 CFG2 寄存器
    ADC0_CFG2&=~(ADC_CFG2_ADACKEN_MASK          //异步时钟输出禁止
            + ADC_CFG2_ADHSC_MASK               //普通转换
            + ADC_CFG2_ADLSTS_MASK);            //默认最长采样时间
    //3.2 选择 b 通道或是 a 通道
    (chnGroup==MUXSEL_B)?(ADC0_CFG2 |=(ADC_CFG2_MUXSEL(1)))
                        :(ADC0_CFG2 &=~(ADC_CFG2_MUXSEL(1)));
    //4.配置 ADC0_SC2: 软件触发,比较功能禁用; DMA 禁用;
    //默认外部参考电压 VREFH/VREFL
    ADC0_SC2 = 0;
    //5.ADC0_SC3 寄存器硬件均值使能,配置硬件滤波次数
    ADC0_SC3 |= (ADC_SC3_ADCO_MASK | ADC_SC3_AVGE_MASK | ADC_SC3_AVGS
((uint_8)HDAve));

    //选择差分输入或是单端输入
    if (AD_DIFF == diff)                        //选择差分输入
    {
        ADC0_SC1A |= (ADC_SC1_DIFF_MASK);
        adc_cal();                              //差分情况,需校验
    }
    else                                        //选择单端输入
    {
        ADC0_SC1A &= ~(ADC_SC1_DIFF_MASK);
    }
    //禁用 ADC 模块中断
    ADC0_SC1A &= ~(ADC_SC1_AIEN_MASK);
}

//==========================================
//函数名称: adc_read
//功能概要: 进行一个通道的一次 AD 转换
//参数说明: channel: 见 MKL25Z128VLK4 芯片 ADC 通道输入表
//==========================================
uint_16 adc_read(uint_8 channel)
{
    uint_16 ADCResult = 0;

    //设置 SC1A 寄存器通道号
    ADC0_SC1A = ADC_SC1_ADCH(channel);

    //等待转换完成
```

```
    while(!(ADC0_SC1A & ADC_SC1_COCO_MASK));

    //读取转换结果
    ADCResult = (uint_16)ADC0_RA;
    //清 ADC 转换完成标志
    ADC0_SC1A &= ~ADC_SC1_COCO_MASK;
    //返回读取结果
    return ADCResult;
}

//---------内部函数-------------------------------------------------------------------
//=====================================================================
//函数名称: adc_cal
//功能概要: adc 模块校正功能函数
//说明: 在校正之前,须正确配置 ADC 时钟、采样时间、模式、硬件滤波 32 次,
//      详见 KL25 芯片手册 28.4.6
//=====================================================================
void adc_cal()
{
  uint_8 cal_var;

  ADC0_SC2 &= ~ADC_SC2_ADTRG_MASK;                              //使能软件触发
  ADC0_SC3 &= ( ~ADC_SC3_ADCO_MASK & ~ADC_SC3_AVGS_MASK);  //单次转换
  ADC0_SC3 |= ( ADC_SC3_AVGE_MASK | ADC_SC3_AVGS(3));       //硬件平均滤波 32 次
  ADC0_SC3 |= ADC_SC3_CAL_MASK;                                 //开始校验
  while (!(ADC0_SC1A & ADC_SC1_COCO_MASK));                     //等待转换完成

  if (ADC0_SC3& ADC_SC3_CALF_MASK) goto adc_cal_exit;           //校正失败
  //校正正确,继续执行
  //计算正向输入校正
  cal_var = 0x00;
  cal_var = ADC0_CLP0;
  cal_var += ADC0_CLP1;
  cal_var += ADC0_CLP2;
  cal_var += ADC0_CLP3;
  cal_var += ADC0_CLP4;
  cal_var += ADC0_CLPS;

  cal_var = cal_var/2;
  cal_var |= 0x8000;                                            //Set MSB
  ADC0_PG = ADC_PG_PG(cal_var);

  //计算负向输入校正
  cal_var = 0x00;
  cal_var = ADC0_CLM0;
  cal_var += ADC0_CLM1;
  cal_var += ADC0_CLM2;
  cal_var += ADC0_CLM3;
  cal_var += ADC0_CLM4;
  cal_var += ADC0_CLMS;
```

```
    cal_var = cal_var/2;
    cal_var |= 0x8000;                                          //Set MSB
    ADC0_MG = ADC_MG_MG(cal_var);
    ADC0_SC3 &= ~ADC_SC3_CAL_MASK ;                             //清 CAL
    adc_cal_exit:
    asm("NOP");
}
```

10.2 数字/模拟转换器 DAC

10.2.1 数/模转换器 DAC 的通用基础知识

1. DA 转换器的通用基本结构

当 MCU 需要把处理后的信息反馈到控制设备上时,就需要把数字量转换成模拟量,完成这种转换的电路称为数/模转换器(Digital-to-Analog Converter, DAC),DA 转换器的工作就是将输入的二进制数字量转换成模拟量,以电压或电流的形式输出。

DA 转换器实质上就是一个译码器(也称为解码器)。一般使用的 DA 转换器为线性的转换器,如式(10-1),其输出的模拟电压 V_0 和输入数字量 D_n 之间成正比关系,其中,V_{REF} 为参考电压。

$$V_0 = D_n \cdot V_{REF} \tag{10-1}$$

图 10-3 中,DA 转换器将输入的每一位二进制代码(D_n)按其权值大小转换成相应的模拟量,然后将代表各位的模拟量相加,则所得的总模拟量就与数字量成正比,如式(10-2)所示。这样,就实现了从数字量到模拟量的转换。

$$D_n = d_{n-1} \cdot 2^{n-1} + d_{n-2} \cdot 2^{n-2} + \cdots + d_1 \cdot 2^1 + d_0 \cdot 2^0 = \sum_{i=0}^{n-1} d_i\, 2^i \tag{10-2}$$

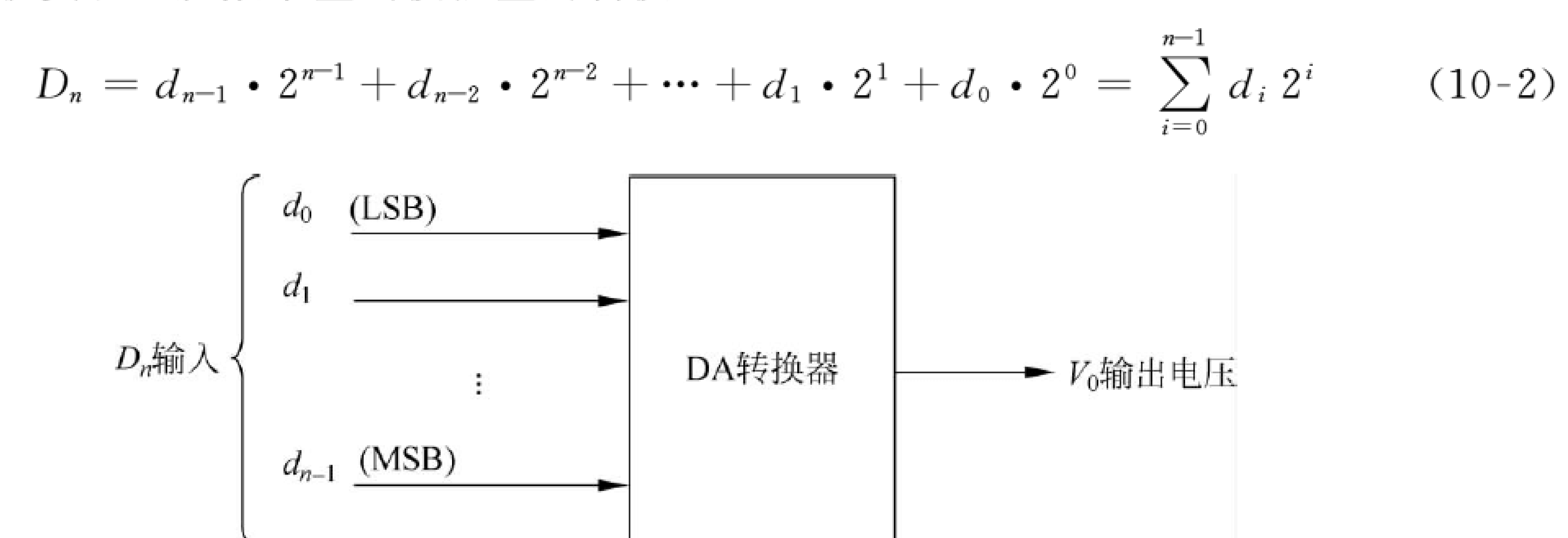


图 10-3 DA 转换器的工作原理

将式(10-2)代入式(10-1)可得式(10-3)。

$$\begin{aligned} V_0 &= d_{n-1} \cdot 2^{n-1} \cdot V_{REF} + d_{n-2} \cdot 2^{n-2} \cdot V_{REF} + \cdots + d_1 \cdot 2^1 \cdot V_{REF} + d_0 \cdot 2^0 \cdot V_{REF} \\ &= \sum_{i=0}^{n-1} d_i \cdot 2^i \cdot V_{REF} \end{aligned} \tag{10-3}$$

由上述公式可知,DA 转换器输出的电压 V_0,等于代码为 1 的各位所对应的各分模拟电压之和。

DA转换器一般由数码缓冲寄存器、模拟电子开关、参考电压、解码网络和求和电路等组成,见图10-4。数字量以串行或并行方式输入,并存储在数码缓冲寄存器中;寄存器输出的每位数码驱动对应数位上的电子开关,将在解码网络中获得的相应数位权值送入求和电路;求和电路将各位权值相加,便得到与数字量对应的模拟量。

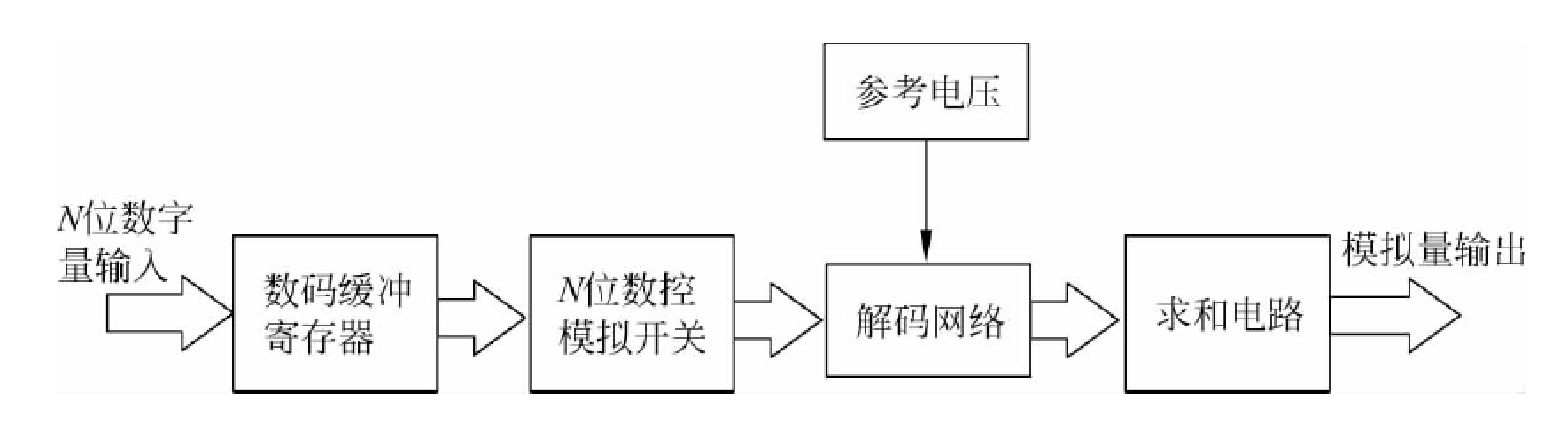


图10-4　N位DA转换器组成框图

2. DA转换器的主要技术指标

1) 分辨率

分辨率用于表征DA转换器对输入微小量变化的敏感程度。分辨率指的就是DA转换器模拟输出电压可能被分离的等级数,MCU就是可用输入数字量的位数n表示DA转换器的分辨率;除此之外,也可用D/A转换器的最小输出电压与最大输出电压之比来表示分辨率,如式(10-4)。分辨率越高,转换时对输入量的微小变化的反应越灵敏。而分辨率与输入数字量的位数有关,n越大,也就是输入的数字量位数越多,DA转换器的分辨率越高。

$$\text{分辨率} = \frac{\Delta U}{U_m} = \frac{1}{2^n - 1} \tag{10-4}$$

2) 转换精度

DA转换器的转换精度是指输出模拟电压的实际值与理想值之差,即最大静态转换误差。

3) 转换速度

在使用DA转换器的时候,其转换速度指的就是从输入的数字量发生突变开始,到输出电压进入与稳定值相差±0.5最低有效位(LSB)范围内所需要的时间,也称为建立时间。目前,单片集成DA转换器(不包括运算放大器)的建立时间最短达到0.1μs以内,即图10-5中建立时间t_{set}表示的时间小于0.1μs。

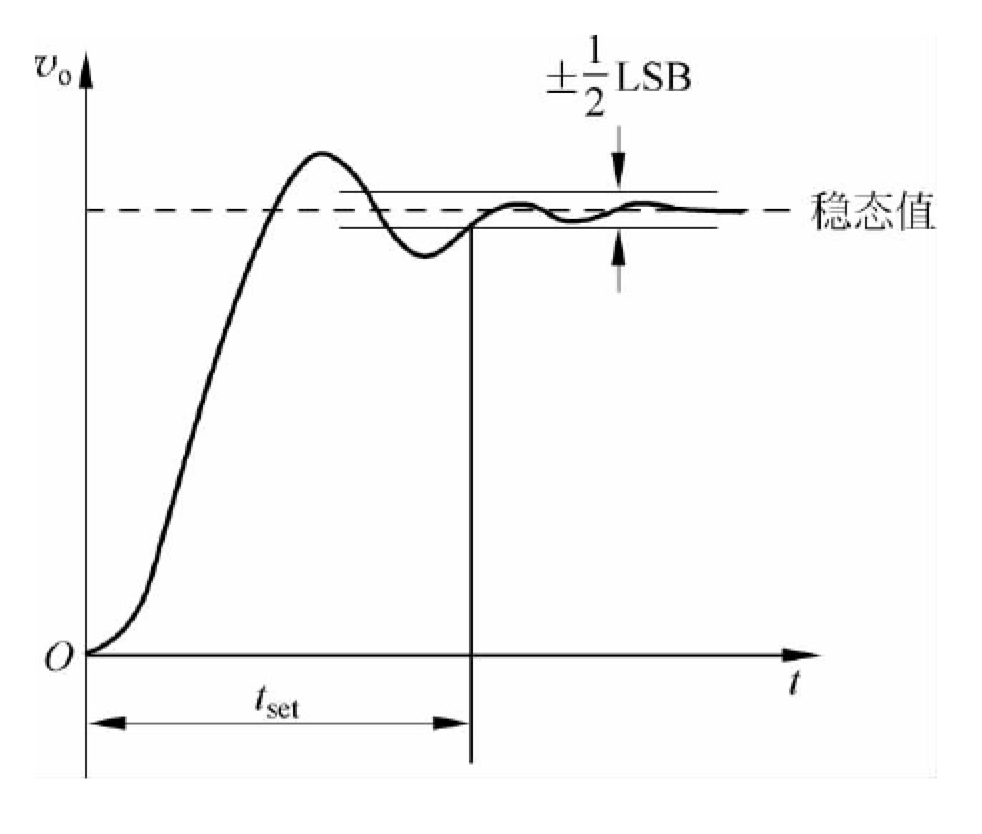


图10-5　DA转换器转换速度演示波形

10.2.2　DAC驱动构件及使用方法

在使用DAC驱动构件之前,需要先对KL25这款芯片所带的DA模块有一个具体的认识。下面从它的结构、特性和工作模式等几个方面来了解一下。

1. KL25芯片DAC模块简介

1) DAC模块结构

DAC模块可以选择两路参考电压:DACREF_1和DACREF_2,其分别连接至V_{REFH}和V_{DDA}。V_{REFH}输出的是一个精准的3.3V电压,V_{DDA}是KL25使用的3.3V工作电压。DAC

模块使用的输出引脚默认为 PTE30，当 DAC 使能时，将会转换 DACDAT[11:0]的值或者把数据缓冲区的值转换成模拟电压。电压转换精度为 Vin/4096，输出电压范围在 Vin/4096～Vin 之间。

2）DAC 模块的特性

DAC 模块的特性：①片内可编程的输出电压产生器（电压输出从 1/4096Vin 到 Vin，Vin 步长是 1/4096Vin）；②Vin 可以从两个参考电源中选择；③在正常停止模式下的待机操作；④支持两个 16 位长的数据缓冲区，由数据寄存器 DAT0 和 DAT1 组成；⑤支持 DMA 操作。

3）DAC 三种工作模式

当 DAC 模块使能，但是缓冲未使能时，DAC 模块会将数据寄存器 DAT0 中的数据转换为模拟电压输出。当 DAC 模块和缓冲均被使能时，DAC 模块会将缓冲区的数据转换为模拟输出电压。当硬件触发或者软件触发发生时，数据缓冲读取指针将向下一个。

DAC 模块缓冲区工作模式可以被配置为正常模式、摆动模式、一次扫描模式。在这些模式下，数据缓冲区的读指针可以设置为任意一个 0 到 DAC 控制寄存器 DAC0_C2 的 DACBFUP 域之间的一个值。

（1）正常模式：缓冲区作为一个循环缓冲区工作，当触发发生时，读指针每次加 1。当读指针到达顶部时，在下次触发事件时回到 0。

（2）摆动模式：当读指针到达顶部时它不返回到 0，而是在下次触发时减 1，直到减到 0 为止。

（3）一次扫描模式：当事件发生时，读指针每次加 1，当到达顶部时停止。复位后读指针返回 0。

KL25 的 DAC 模块缓冲区工作模式，只有正常模式和一次扫描模式。

2. DAC 引脚

在本书使用的 KL25 封装中，DA 模块仅有一个对外引脚 PTE30。与 AD 模块相比，KL25 提供的 DA 对外引脚很少。主要原因是，在大部分情况下，可以使用 PWM 来实现对外的不同电压输出，而少部分需要稳定直流电源的电器才会用到 DA 模块。在本书使用的 KL25 芯片中，提供了 35 路可以复用成 PWM 功能的对外引脚，足够一般情况下的使用。具体 PWM 使用方法，请参考 7.2 节的内容。

3. DAC 驱动构件基本要点分析

DA 模块具有初始化、执行 DAC 转换两个操作。按照构件的思想，可将它们封装成独立的功能函数。DA 构件包括 dac.h 和 dac.c 文件。DA 构件头文件中主要包括相关宏定义、DA 的功能函数原型说明等内容。DA 构件程序文件的内容是给出 DA 各功能函数的实现过程。

1）模块初始化（void dac_init）

```
void dac_init(uint_8 RefVoltage);
```

初始化 DAC 模块要配置 DAC0_C0 寄存器，选择参考电压，设置软件触发、配置 DAC0_C1

寄存器来禁用DMA。需要注意的是,D/A模块对外仅有一个引脚PTE30。

2) DA转换(void dac_convert)

```
void dac_convert(uint_16 data);
```

初始化完成后,使用dac_convert()函数通过data参数来设置DAC0或者DAC1缓冲区的值,值大小在0~4095之间,在DA模块使能时,DA模块会将缓冲区中的值转换为对应的模拟电压。

4. DAC驱动构件头文件

```
//==========================================================
//文件名称: dac.h
//功能概要: dac底层驱动构件头文件
//版权所有: 苏州大学恩智浦嵌入式中心(sumcu.suda.edu.cn)
//更新记录: 2013-04-07    V1.0
//==========================================================

#ifndef _DAC_H                              //防止重复定义(开头)
#define _DAC_H

#include "common.h"                         //包含公共要素头文件

#define DAC_VREFH   0
#define DAC_VDDA    1

//==========================================================
//函数名称: dac_init
//函数返回: 无
//参数说明: RefVoltage: 参考电压选择DAC_VREFH或DAC_VDDA
//功能概要: 初始化DAC模块设定
//==========================================================
void dac_init(uint_8 RefVoltage);

//==========================================================
//函数名称: dac_convert
//参数说明: data: 需要转换成模拟量的数字量,范围(0~4095)
//功能概要: 执行DAC转换
//==========================================================
void dac_convert(uint_16 data);

#endif
```

5. DAC驱动构件使用方法

DAC驱动构件的头文件(dac.h)中包含的内容有:给出两个对外服务函数的接口说明及声明,函数包括DAC初始化函数(dac_init)、执行DAC转换函数(dac_convert)。

下面以制作一个基于KL25的DAC模块的呼吸灯为例,介绍DAC构件的使用方法。步骤如下。

(1) 初始化 DAC 模块,选择参考电压 VDDA=3.3V:

```
dac_init(DAC_VDDA);
```

(2) 在主循环中,使用 DAC 对数字量 VReference 进行转换,VReference 的取值范围在 2200～2600 之间。(根据不同型号的二极管,这个取值范围可能有所变动,读者可根据实际情况,在 0～4095 之间自行更改其值。)

```
dac_convert(VReference);
```

(3) 将一个发光二极管的正极接在 KL25 评估板上的 PTE30 号脚上,将其负极接在评估板上任意一个 GND 上,即可看到小灯的呼吸效果。

6. DAC 驱动构件测试实例

DAC 测试实例工程位于网上教学资源中的“..\ch10-ADC-DAC-CMP \KL25-DAC”文件夹,该样例演示 DAC 将数字量 2200～2600 转换为对应电压,可将一个 LED 的正极接在 PTE30 上,将负极接在 GND 上,可观察到发光二极管的效果。

DAC 测试主函数文件 main.c 代码如下。

```
//说明见工程文件夹下的 Doc 文件夹内 Readme.txt 文件
//==========================================

#include "includes.h"                              //包含总头文件

int main(void)
{
    //1. 声明主函数使用的变量
    uint_32   mRuncount;                           //主循环计数器
    uint_16   VReference;
    uint_8 light_flag;
    //2. 关总中断
    DISABLE_INTERRUPTS;

    //3. 初始化外设模块
    light_init(RUN_LIGHT_BLUE, LIGHT_ON);          //蓝灯初始化
    uart_init(UART_1, 9600);                       //使能串口 1,波特率为 9600
    uart_init(UART_2, 9600);                       //使能串口 2,波特率为 9600
    uart_send_string(UART_1, "This is DAC Test!\r\n");  //串口发送初始化提示
    dac_init(DAC_VDDA);                          //DAC 初始化,选择参考电压 VDDA=3.3V
    //4. 给有关变量赋初值
    mRuncount=0;                                   //主循环计数器
    VReference=2000;                               //DAC 参考数字量
    light_flag=1;
    //5. 使能模块中断
    uart_enable_re_int(UART_1);                    //使能串口 1 接收中断
    uart_enable_re_int(UART_2);                    //使能串口 2 接收中断
    //6. 开总中断
    ENABLE_INTERRUPTS;
```

```
        //进入主循环
        //主循环开始==================================
        for(;;)
        {
            //运行指示灯(RUN_LIGHT)闪烁--------------------------------------------------
            mRuncount++;                              //主循环次数计数器+1
            if (mRuncount >= RUN_COUNTER_MAX)  //主循环次数计数器大于设定的宏常数
            {
                mRuncount=0;                          //主循环次数计数器清零
                //蓝色运行指示灯(RUN_LIGHT_BLUE)状态变化
                light_change(RUN_LIGHT_BLUE);
                //DAC 数字量转换,输出 VReference 的值对应的电压值
                dac_convert(VReference);

                //根据标志位,设置小灯慢慢点亮或慢慢熄灭
                if(light_flag==1)   VReference += 1;
                else if(light_flag==0)   VReference -= 1;

                //VReference 限幅,并反转灯点亮或熄灭的标志位
                if(VReference >= 2600)   light_flag=0;
                if(VReference <= 2200)   light_flag=1;
            }
            //以下加入用户程序--------------------------------------------------------
        }//主循环 end_for
        //主循环结束==================================
    }
```

10.2.3　DAC 驱动构件的编程结构

本节主要介绍如何根据DAC模块的各个寄存器的功能,结合上文给出的dac.h编写具体的DAC的驱动。

KL25的DAC转换模块有8个8位寄存器,包括一个DAC状态控制寄存器(DAC0_SR),三个DAC控制寄存器DAC0_C0、DAC0_C1和DAC0_C2,4个DAC数据寄存器DAC0_DAT0L和DAC0_DAT0H、DAC0_DAT1L和DAC0_DAT1H。通过对这些寄存器的编程,就可以获取DAC的转换数据。

1. DAC状态寄存器

D7~D2(Reserved)——该位段保留且只读为0。

D1(DACBFRPTF)——DAC缓冲读指针的顶部标志。0表示DAC缓冲区读指针不等于0;1表示DAC缓冲区读指针等于0。

D0(DACBFRPBF)——DAC缓冲读指针的底部标志。0表示DAC缓冲区读指针不等于C2[DACBUFUP];1表示DAC缓冲区读指针等于C2[DACBUFUP]。

2. DAC控制寄存器

控制寄存器DAC0_C0具有使能DAC、硬件/软件触发选择、功耗选择和DAC模块的参考电压选择等功能,其结构如表10-14所示。复位后,各位均为0。

表 10-14　DAC0_C0 结构

位	D7	D6	D5	D4	D3	D2	D1	D0
读	DACEN	DACRFS	DACTRGSEL	0	LPEN	0	DACBTIEN	DACBBIEN
写				DACBWTRG				

D7(DACEN)——DAC 使能位。0 表示 DAC 模块禁用,1 表示 DAC 模块使能。

D6(DACRFS)——DAC 电压参考选择位。0 表示选择 DACREF_1 作为参考电压,1 表示选择 DACREF_2 作为参考电压。

D5(DACTRGSEL)——DAC 模块触发方式选择位。0 表示 DAC 硬件触发,1 表示 DAC 软件触发。

D4(DACSWTRG)——DAC 软件触发位。0 表示 DAC 软件触发禁止,1 表示 DAC 软件触发使能。该位读为 0。如果 DAC 选择软件触发且使能缓冲,那么写 1 则会使缓冲区读指针向前加 1。

D3(LPEN)——DAC 低功耗控制位。0 表示高功耗模式,1 表示低功耗模式。

D2(Reserved)——该位保留且只读为 0。

D1(DACBTIEN)——DAC 缓冲区读指针顶部标志中断使能位。0 表示 DAC 缓冲区读指针顶部标志中断禁止,1 表示 DAC 缓冲区读指针顶部标志中断使能。

D0(DACBBIEN)——DAC 缓冲区读指针底部标志中断使能位。0 表示 DAC 缓冲区读指针底部标志中断禁止,1 表示 DAC 缓冲区读指针底部标志中断使能。

3. DAC 控制寄存器

本寄存器主要有 DMA 的使能和 DAC 缓冲区相关设置等功能,其结构如表 10-15 所示。复位后,各位均为 0。

表 10-15　DAC0_C1 结构

数据位	D7	D6～D3	D2	D1	D0
读	DMAEN	0	DACBFMD	0	DACBFEN
写					

D7(DMAEN)——DMA 控制使能位。0 表示 DMA 禁止,1 表示 DMA 使能。当 DMA 使能时,DMA 请求由原始中断产生,并且此时模块的中断不会发生。

D6～D3(Reserved)——该位段保留且只读为 0。

D2(DACBFMD)——DAC 缓冲区工作模式选择位。0 表示正常模式,1 表示单次扫描模式。

D1(Reserved)——该位保留且只读为 0。

D0(DACBFEN)——DAC 缓冲区使能位。0 表示缓冲区读指针禁止,转换的数据总是缓冲第一个字长的数据。1 表示缓冲区读指针使能,转换的数据总是读指针指向的字,这意味着转换的数据来自于缓冲区任何字长的数据。

4. DAC 控制寄存器

本寄存器主要设置 DAC 缓冲区的读指针和缓冲区上限,其结构如表 10-16 所示。

表 10-16　DAC0_C2 结构

数据位	D7～D5	D4	D3～D1	D0
读	0	DACBFRP	0	DACBFUP
写				
复位	0			1

D7～D5(Reserved)——该位段保留且只读为0。

D4(DACBFRP)——DAC缓冲区读指针。该位保存了缓冲区读指针的当前值。

D3～D1(Reserved)——该位段保留且只读为0。

D0(DACBFUP)——DAC缓冲区上限。该位选择DAC缓冲区的上限,缓冲区读指针不能超过此上限。该位为0表示缓冲区只有一个16位数据(DAT0),该位为1表示缓冲区只有两个16位数据(DAT0,DAT1)。

5. DAC 数据寄存器

数据寄存器DAT0由高位寄存器ADC0_ADT0H和低位寄存器ADC0_ADT0L组成。当DAC缓冲区禁用时,DAT0[11:0]控制输出电压,计算公式为 Vout = Vin×(1+DAT0[11:0])/4096;当DAC缓存区使能时,DAT0被映射到2字的数据缓冲区中。

6. DAC 数据寄存器

数据寄存器DAT1由高位寄存器ADC0_ADT1H和低位寄存器ADC0_ADT1L组成。当DAC缓冲区禁用时,DAT1不被使用;当DAC缓存区使能时,DAT0被映射到2字的数据缓冲区中。

10.2.4　DAC 驱动构件的设计

本节主要介绍如何根据DAC模块的各个寄存器的功能,结合上文给出的dac.h编写具体的DAC的驱动。

1. KL 系列 MCU 的 DAC 模块功能概述

DAC具有初始化和DAC转换两种基本操作。按照构件的思想,可将它们封装成两个独立的功能函数,初始化函数完成对DAC模块的工作属性的设定,DAC转换函数则完成实际的转换任务。对DAC模块进行编程,实际上已经涉及对硬件底层寄存器的直接操作,因此,可将初始化和DAC转换两种基本操作所对应的功能函数共同放置在命名为dac.c的文件中,并按照相对严格的构件设计原则对其进行封装,同时配以命名为dac.h的头文件,用来定义模块的基本信息和对外接口。

按照模块所具有的基本操作来确定构件中应该具有哪些功能集合,是很自然也很重要的事情。但是,要实现编程的构件化,对具体的函数原型的设计则是重中之重。函数原型设计的好坏直接影响构件化编程的成败。下面就以DAC的初始化和DAC转换两种基本操作为例,来说明实现构件化编程的全过程。

2. DAC 驱动构件源码

```
//==========================================================
//文件名称：dac.c
```

```
//功能概要：KL25 DAC 底层驱动程序文件
//版权所有：苏州大学恩智浦嵌入式中心(sumcu.suda.edu.cn)
//更新记录：2013-04-07    V1.0
//==========================================================================

#include "dac.h"   //包含 DAC 驱动程序头文件

//====================内部函数声明====================
//==========================================================================
//函数名称：dac_set_buffer
//函数返回：设置的缓冲区大小值
//参数说明：dacx_base_ptr：DACx 基指针
//          dacindex 缓冲区号：0,1
//          buffval 缓冲区值:0～4095
//功能概要：设置 DACx 缓冲区
//==========================================================================
int dac_set_buffer(DAC_MemMapPtr dacx_base_ptr, uint_8 dacindex, int buffval);


//==========================================================================
//函数名称：dac_init
//函数返回：无
//参数说明：RefVoltage：参考电压选择 DAC_VREFH 或 DAC_VDDA
//功能概要：初始化 DAC 模块设定
//==========================================================================
void dac_init(uint_8 RefVoltage)
{
    SIM_SCGC6 |=SIM_SCGC6_DAC0_MASK;                  //使能 DAC0 时钟

    //配置 DAC0_C0 寄存器
    if(1== RefVoltage)
    DAC0_C0 |= DAC_C0_DACRFS_MASK;                    //选择 VDDA 参考电压 3.3V
    else if(0 == RefVoltage)
    DAC0_C0 &= ~DAC_C0_DACRFS_MASK;                   //选择 VREFH 参考电压 3.3V
    DAC0_C0 |= DAC_C0_DACTRGSEL_MASK;                 //软件触发
    //软件触发无效,高功耗模式,缓冲区置底中断禁止,缓冲区置顶中断禁止
    DAC0_C0 &= ~(DAC_C0_DACSWTRG_MASK | DAC_C0_LPEN_MASK
                        | DAC_C0_DACBBIEN_MASK | DAC_C0_DACBTIEN_MASK);

   //配置 DAC0_C1 寄存器
   //DMA 禁用,正常工作模式
    DAC0_C1 &= ~(DAC_C1_DMAEN_MASK | DAC_C1_DACBFEN_MASK
                              | DAC_C1_DACBFMD_MASK);
    //使能 DAC0 模块
    DAC0_C0 |= DAC_C0_DACEN_MASK;
}

//==========================================================================
```

```
//函数名称: dac_convert
//参数说明: data: 需要转换成模拟量的数字量,范围(0～4095)
//功能概要: 执行 DAC 转换
//==========================================================
void dac_convert(uint_16 data)
{
    dac_set_buffer(DAC0_BASE_PTR, 0, data);
}

//====================内部函数====================
//==========================================================
//函数名称: dac_set_buffer
//函数返回: 设置的缓冲区大小值
//参数说明: dacx_base_ptr: DACx 基指针
//          dacindex 缓冲区号: 0,1
//          buffval 缓冲区值:0～4095
//功能概要: 设置 DACx 缓冲区
//==========================================================
int dac_set_buffer(DAC_MemMapPtr dacx_base_ptr, uint_8 dacindex, int buffval)
{
    int temp = 0;
    //设置缓冲区低字节
    DAC_DATL_REG(dacx_base_ptr, dacindex)= (buffval&0x0ff);
    //设置缓冲区高字节
    DAC_DATH_REG(dacx_base_ptr, dacindex)= (buffval&0xf00) >>8;
    temp =(DAC_DATL_REG(dacx_base_ptr, dacindex)|
        (DAC_DATH_REG(dacx_base_ptr, dacindex)<<8));
    return temp;
}
```

10.3　比较器 CMP

10.3.1　比较器 CMP 的通用基础知识

1. 电压比较器的作用

比较器模块可以比较两路模拟电压。很多场合需要检测模拟电压,比如一个湿度报警器,传感器模拟信号经过放大后直接与比较器输入端连接,跟参考电压比较,当大小发生变化时,就可以产生中断,实现可控的输出结果。比较器用作模拟电路和数字电路的接口,还可以用作波形产生和变换电路等。利用简单电压比较器可将正弦波变为同频率的方波或矩形波。

2. 比较器的分类

(1) 模拟比较器:将模拟量与一标准值进行比较,当高于该值时,输出高(或低)电平。反之,则输出低(或高)电平。例如,将一温度信号接于运放的同相端,反相端接一电压基准(代表某一温度),当温度高于基准值时,运放输出高电平,控制加热器关闭,反之当温度信号

低于基准值时，运放输出低电平，将加热器接通，这一运放就是一个简单的比较器。

（2）数字比较器：用来比较两组二进制数是否相同，相同时输出（或低）高电平，反之，则输出相反的电平。最简单的数字比较器是一位二进制数比较器，是一个异或门。

10.3.2　CMP 驱动构件及使用方法

1. CMP 引脚

模拟多路复用器（ANMUX）可以从 8 路通道中选择一路模拟信号作为输入信号。6 位的 DAC 可以提供一个信号。MUX 电路可以在整个电压范围内进行操作。表 10-17 给出了 KL25 的 CMP 引脚配置表。

表 10-17　CMP 引脚配置

引　脚　号	引　脚　名	功能 0	功能 5	功能 6
1	PTE0		CMP0_OUT	
21	PTE29	CMP0_IN5		
22	PTE30	CMP0_IN4		
55	PTC0		CMP0_OUT	
62	PTC5			CMP0_OUT
63	PTC6	CMP0_IN0		
64	PTC7	CMP0_IN1		
65	PTC8	CMP0_IN2		
66	PTC9	CMP0_IN3		

注：CMP0_IN6 为 Bandgap 电压，CMP0_IN7 为 DAC 转换电压。

2. CMP 驱动构件基本要点分析

CMP 具有模块初始化、设置 DAC 的值、中断使能、中断除能等基本操作。按照构件的思想，可将它们封装成 4 个独立的功能函数，初始化函数完成对 CMP 模块的工作属性的设定；进行 DAC 值的设置等。CMP 构件头文件中主要包括相关宏定义、CMP 的功能函数原型说明等内容。CMP 构件程序文件的内容是给出 CMP 各功能函数的实现过程。

1）模块初始化（cmp_init）

```
void cmp_init(uint_8 reference,uint_8 plusChannel,uint_8 minusChannel);
```

CMP 初始化函数，主要完成对 CMP 模块工作的参数设定，包括工作时钟、正负通道选择、参考电压选择，还有中断使能等一些基本设置。

2）设置 DAC 的值（dac_set_value）

```
void dac_set_value(uint_8 value);
```

设置 6 位 DAC 输出的值。从 64 个不同等级中选择输出电压，DAC 输出电压＝（Vin/64）×（VOSEL[5:0]＋1），输出电压范围是 Vin/64～Vin。

3）中断使能（cmp_enable_int）

```
void cmp_enable_int();
```

CMP 中断使能。注册 CMP 的中断号(KL25 芯片中断号为 16)。

4) 中断除能(cmp_disable_int)

```
void cmp_disable_int();
```

关闭 CMP 中断使能。

3. CMP 驱动构件头文件

```
//===============================================================
//文件名称: cmp.h
//功能概要: KL25 比较器底层驱动程序头文件
//版权所有: 苏州大学恩智浦嵌入式中心(sumcu.suda.edu.cn)
//版本更新: 2012-11-25   V1.0    初始版本
//===============================================================

#ifndef   _HSCMP_H                  //防止重复定义(开头)
#define   _HSCMP_H

#include "common.h"                 //包含公共要素头文件

//===============================================================
//函数名称: cmp_init
//函数返回: 无
//参数说明:reference:参考电压选择   0=Vin1in   1=Vin2in
//         plusChannel: 正比较通道号
//         minusChannel: 负比较通道号
//通道号 0,1,2,3,4,5 对应引脚 PTC6,PTC7,PTC8,PTC9,PTC30,PTC29;
//通道 6 Bandgap;通道 7 DAC 输入
//功能概要: CMP 模块初始化
//===============================================================
void cmp_init(uint_8 reference,uint_8 plusChannel,uint_8 minusChannel);

//===============================================================
//函数名称: dac_set_value
//函数返回: 无
//参数说明: value: dac 输出的转换值
//功能概要: 设置 DAC 输出值
//===============================================================
void dac_set_value(uint_8 value);

//===============================================================
//函数名称: cmp_enable_int
//函数返回: 无
//参数说明: 无
//功能概要: 开比较中断
//===============================================================
void cmp_enable_int();

//===============================================================
```

```
//函数名称：cmp_disable_int
//函数返回：无
//参数说明：无
//功能概要：关比较中断
//==================================================================
void cmp_disable_int();

#endif
```

4. CMP 驱动构件使用方法

在 CMP 驱动构件的头文件(cmp.h)中包含的内容有：给出 4 个对外服务函数的接口说明及声明，函数包括 cmp 初始化函数(cmp_init)、设置 DAC 的值函数(dac_set_value)、开比较中断函数(cmp_enable_int)和关比较中断函数(cmp_disable_int)。

下面以比较模块引脚 PTC7 和 DAC 的模拟输出值为例，介绍构件的使用方法。步骤如下。

(1) 初始化 CMP0 模块，DAC 参考电压 Vin1in，正向通道 0，负向通道 7；

```
cmp_init(0,1,7);
```

(2) 设置 DAC 的值，每次让 dac_value 的值自加，并设置 6 位 DAC 输出；

```
dac_set_value(dac_value);     //6 位 DAC 设置输出
```

(3) 使能 CMP 模块中断；

```
cmp_enable_int();
```

(4) 初始化 PTD1，在主循环中定时反转其输出；

(5) 将 PTD1 接 PTC7；

(6) 通过串口调试工具读取比较器的输出。

5. CMP 驱动构件测试实例

测试工程功能概述如下。

(1) 串口通信格式：波特率 9600，1 位停止位，无校验。

(2) 上电时，调试串口输出"This is CMP Test!"。

(3) 主循环中，改变 RUN_LIGHT_BLUE 小灯状态。

(4) 程序中将 PTC7 的值与 6 位 DAC 的值进行比较，内部 6 位 DAC 将 0～60 数字量转换成 0～3V 电压输入给负向通道，正向通道输入电压 0 或 3.3V，CMP 比较两通道电压值，根据比较结果在串口输出提示。

(5) PC 向 MCU 发送数据时，MCU 进入串口接收中断。

CMP 构件的测试工程位于网上教学资源中的"..\ch10-ADC-DAC-CMP\KL25-CMP"文件夹。

CMP 测试主函数文件 main.c 代码如下。

```
//说明见工程文件夹下的 Doc 文件夹内 Readme.txt 文件
//==================================================================
```

```
#include "includes.h"                          //包含总头文件

int main(void)
{
    //1. 声明主函数使用的变量
    uint_32  mRuncount;                        //主循环计数器
    uint_8 dac_value;
    //2. 关总中断
    DISABLE_INTERRUPTS;

    //3. 初始化外设模块
    light_init(RUN_LIGHT_BLUE, LIGHT_ON);   //蓝灯初始化
    gpio_init(PORT_D|1, 1, 0);                 //初始化 PTD1
    uart_init(UART_1, 9600);                   //使能串口 1,波特率为 9600
    uart_init(UART_2, 9600);                   //使能串口 2,波特率为 9600
    uart_send_string(UART_1, "This is CMP Test!\r\n");  //串口发送初始化提示
    cmp_init(0,1,7);          //初始化 CMP0 模块,DAC 参考电压 Vin1in,正向通道 0,负向通道 7
    //4. 给有关变量赋初值
    mRuncount=0;                               //主循环计数器
    dac_value=0;
    //5. 使能模块中断
    cmp_enable_int();                          //使能 CMP0 中断和串口 1 作为通信
    uart_enable_re_int(UART_1);                //使能串口 1 接收中断
    uart_enable_re_int(UART_2);                //使能串口 2 接收中断
    //6. 开总中断
    ENABLE_INTERRUPTS;
    //进入主循环
    //主循环开始==================================
    for(;;)
    {
        //运行指示灯(RUN_LIGHT)闪烁---------------------------------------------
        mRuncount++;                           //主循环次数计数器+1
        if (mRuncount >= RUN_COUNTER_MAX)  //主循环次数计数器大于设定的宏常数
        {
            mRuncount=0;                       //主循环次数计数器清零
            //蓝色运行指示灯(RUN_LIGHT_BLUE)状态变化
            light_change(RUN_LIGHT_BLUE);
            gpio_reverse(PORT_D|1);            //反转 D 口 1 号引脚输出,制造 CMP 读取变化
            dac_value += 10;                   //DAC 输出值增加
            dac_set_value(dac_value);          //6 位 DAC 设置输出
            //6 位 DAC 输出值清零,值最大为 63
            //这里因为上面 DAC 值累加幅度为 10,所以这里设置 60
            if(dac_value>60)   dac_value=0;
        }
        //以下加入用户程序--------------------------------------------------------
    }//主循环 end_for
    //主循环结束==================================
}
```

10.3.3　CMP 驱动构件的编程结构

1. 相关名词解释

正向输入：比较器的正向输入值，可以是参考电压，也可以是 6 位 DAC 的值。

负向输入：比较器的负向输入值，可以是参考电压，也可以是 6 位 DAC 的值。

敏感模式：比较器对哪种类型的电压比较敏感，共有 4 种敏感模式。

2. CMP 控制寄存器 0

本寄存器可设置 CMP 模块的滤波采样次数和滞环控制等功能，其结构如表 10-18 所示。复位后，各位均为 0。

表 10-18　CMPx_CR0 结构

位	D7	D6～D4	D3、D2	D1、D0
读	0	FILTER_CNT	0	HYSTCTR
写				

D7(0)——保留位，只读，且各位值为 0。

D6～D4(FILTER_CNT)——滤波采样次数，在接受一个新的状态之前，必须与之前比较器输出滤波器一致。000 表示禁止滤波，若 SE=1，COUT 位为 0(这不是一个合法位，不建议使用)；001 表示采样一次，比较器输出的是简单采样；010 表示连续两次采样；011 表示连续三次采样；100 表示连续 4 次采样；101 表示连续 5 次采样；110 表示连续 6 次采样；111 表示连续 7 次采样。编程时注意：只有在设置了 CR1[SE]=0 和 FILTER_CNT=0x00 后，才可以对采样滤波器次数进行设置。

D3、D2(0)——保留位。只读，且各位值为 0。

D1、D0(HYSTCTR)——比较器滞环控制。定义了可编程滞环控制等级。这个滞环值与设备定义的每个等级有关。准确值需要查看 KL25 数据手册。00 表示 Level 0，01 表示 Level 1，10 表示 Level 2，11 表示 Level 3。

3. CMP 控制寄存器 1

本寄存器可设置 CMP 模块的采样使能、触发模式、比较输出的相关设置等功能，其结构如表 10-19 所示。复位后，各位均为 0。

表 10-19　CMPx_CR1 结构

位	D7	D6	D5	D4	D3	D2	D1	D0
读	SE	WE	TRIGM	PMODE	INV	COS	OPE	EN
写								

D7(SE)——采样使能位。0 表示禁止外部时钟控制采样，1 表示采样使能。SE 和 WE 可以在任何时候设置，但是两个同时设置 SE 和 WE 都会被清 0，所以要避免两位同时写 1。

D6(WE)——窗口功能使能位。0 表示禁止窗口功能，1 表示窗口功能使能。

D5(TRIGM)——触发模式使能。当 TRIGM=1 时，CMP 和 DAC 被配置为 CMP 触发模式，此时 CMP 应当被启用。如果 DAC 是被用来作为一个参考的 CMP，它也应当被使能。

D4(PMODE)——CMP 触发模式依赖于外部定时器,它会定期使能 CMP 和 6 位 DAC,以产生一个触发比较。

D3(INV)——比较输出翻转。该位允许选择模拟比较器的极性。当 CR1[OPE]=0 时,也显示在 COUT 输出上。0 表示比较输出不翻转,1 表示比较输出翻转。

D2(COS)——比较输出选择位。0 表示设置过滤比较输出等于 COUT,1 表示设置非过滤比较输出等于 COUTA。

D1(OPE)——比较输出引脚使能位。0 表示比较器输出(CMP0)在相关的 CMP0 输出引脚上不可用。如果比较器没有这个引脚,该位不受影响。1 表示比较器输出(CMP0)在相关的 CMP0 输出引脚上可用。

D0(EN)——比较器模块使能。EN 位使能模拟比较器模块。当模块不使能时,它处于关闭状态,不消耗电源,当用户从模拟多路复用器到正负端口进行相同的输入时,模块将自动禁用。0 表示模拟比较器禁止,1 表示模拟比较器使能。

4. CMP 滤波周期寄存器

D7～D0(FILT_PER)——滤波采样周期。当 CR1[SE]等于 0 时,这些位确定采样周期。设置 FILT_PER=0x0,禁止滤波。当 CR1[SE]=1 时,这些位无效。在这种情况下,外部采样信号来决定采样周期。

5. CMP 状态控制寄存器

本寄存器可设置 CMP 模块的 DMA 使能控制、比较器的中断方式使能等功能,其结构如表 10-20 所示。复位后,各位均为 0。

表 10-20　CMPx_SCR 结构

位	D7	D6	D5	D4	D3	D2	D1	D0
读	0	DMAEN	0	IER	IEF	CFR	CFF	COUT
写						w1c	w1c	

D7(0)——保留位。只读,且各位值为 0。

D6(DMAEN)——DMA 使能控制位。DMAEN 用来使能由 CMP 触发的 DMA 转换。当 CFR 或 CFF 置位时,当 DMAEN 置位时,DMA 使能。0 表示 DMA 禁止,1 表示 DMA 使能。

D5(0)——保留位。只读,且各位值为 0。

D4(IER)——比较器上升沿中断使能位。IER 位使能来自 CMP 的 CFR 中断。当 IER 被置位时,且 CFR 被置位时,中断将会被使能。0 表示禁止中断,1 表示中断使能。

D3(IEF)——下降沿中断使能位。IEF 位使能来自 CMP 的 CFF 中断。当 CFF 被置位时,且 IEF 位置位,中断将会被使能。0 表示禁止中断,1 表示中断使能。

D2(CFR)——模拟比较上升标志。在正常操作模式中,在 COUT 上检测到上升沿跳变时,CFR 置位。该位写 1 清 0。在停止模式期间,CFR 仍对电平敏感。0 表示在 COUT 的上升沿不检测,1 表示 COUT 产生在上升沿。

D1(CFF)——模拟比较器下降标志。在正常操作模式中,在 COUT 上检测到下降沿跳变时,CFF 置位。该位写 1 清 0。在停止模式期间时,CFF 对电平敏感。0 表示在 COUT 的下降沿不检测,1 表示 COUT 产生在下降沿。

D0(COUT)——该位返回比较器此输出的值。当比较器模块禁止(即 CR1[EN]＝0)，这一位将会复位为 0，读作 CR1[INV]。写操作对该位无效。

6. DAC 控制寄存器

本寄存器可设置 CMP 模块中的 DAC 功能使能、参考电压源选择和输出电压选择等功能，其结构如表 10-21 所示。复位后，各位均为 0。

表 10-21　CMPx_DACCR 结构

位	D7	D6	D5～D0
读	DACEN	VRSEL	VOSEL
写			

D7(DACEN)——DAC 功能使能位。当 DAC 使能禁止时，它会关闭以节省电力。0 表示禁止 DAC 功能，1 表示 DAC 功能使能。

D6(VRSEL)——电源电压基准源选择。0 表示选择梯行电阻网络参考电压 Vin1，1 表示选择梯行电阻网络参考电压 Vin2。

D5～D0(VOSEL)——DAC 输出电压选择。从 64 个不同等级中选择输出电压，DACO＝(Vin/64)×(VOSEL[5:0]＋1)，DACO 电压范围是 Vin/64～Vin。

7. 多路选择控制寄存器

本寄存器可设置 CMP 模块中模式使能与正负输入源的选择等功能，其结构如表 10-22 所示。复位后，各位均为 0。

表 10-22　CMPx_MUXCR 结构

位	D7	D6	D5～D3	D2～D0
读	PSTM	0	PSEL	MSEL
写				

D7(PSTM)——模式使能位。0 表示通过模式禁止，1 表示通过模式使能。此位用来通过模式启用 MUX。通常模式总是可用的，但对于某些设备，由于缺乏封装引脚，此功能必须被禁止。

D6(0)——保留位。只读，且各位值为 0。

D5～D3(PSEL)——这些位决定哪一个输入被选择为比较器的正输入，INx 输入的详情请参照 CMP、DAC 和 ANMUX 结构图。当一个不合适的操作为复用器选择了相同输入时，比较器将自动关闭，阻止自身变成一个噪声产生器。000 表示 IN0，001 表示 IN1，010 表示 IN2，011 表示 IN3，100 表示 IN4，101 表示 IN5，110 表示 IN6，111 表示 IN7。

D2～D0(MSEL)——这些问题决定哪一个输入被选择为比较器的负输入，INx 输入的详情请参照 CMP、DAC 和 ANMUX 结构图。当一个不合适的操作为复用器选择了相同输入时，比较器将自动关闭，阻止自身变成一个噪声产生器。000 表示 IN0，001 表示 IN1，010 表示 IN2，011 表示 IN3，100 表示 IN4，101 表示 IN5，110 表示 IN6，111 表示 IN7。

10.3.4　CMP 驱动构件的设计

本节主要介绍如何根据 CMP 模块的各个寄存器的功能，结合上文给出的 cmp.h 编写

具体的 CMP 的驱动。

1. KL 系列 MCU 的 CMP 模块功能概述

KL25 的 CMP 比较器电路使用与电源相同的电压范围,是在整个电源电压范围内比较;通过编程来控制滞环;并且可以选择上升沿中断,下降沿中断或沿跳变中断;具有着广泛的输出能力。CMP 的比较器功能主要是配合一个 6 位的 64-tap 精度的 DAC 的模拟输出电压,和模拟复用器来做比较,输出的结果可以进行处理以及滤波。

1) CMP 原理图

KL25 的比较器模块原理图如图 10-6 所示,有两个模拟复用器(MUX),MUX 可以从 8 路通道中选择一路模拟信号作为输入信号。内部的 6 位 DAC 也可以提供一个电压信号。MUX 电路可以在整个电源电压范围内进行操作。6 位 DAC 是一个 64-tap 的电阻型阶梯

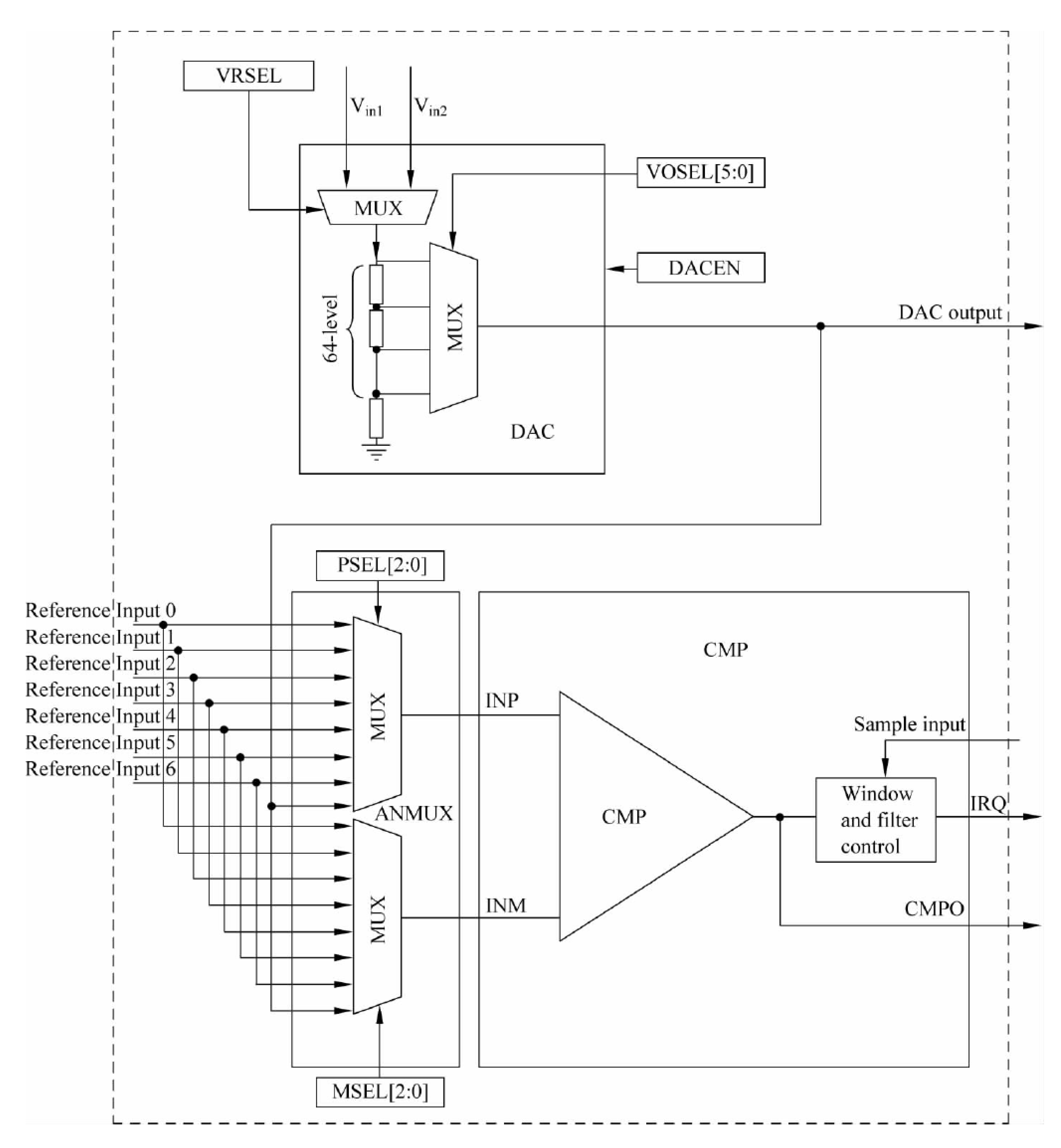


图 10-6 CMP、DAC 和 ANMUX 模块框图

网络，它可以为应用程序提供一个可选的参考电压。它可以将电源电压分成 64 个不同的参考电压值。每 6 位数值对应一个输出电压，它的值的范围从 Vin 到 Vin/64。DAC 的参考电压 Vin 可以从 Vin1 和 Vin2 中选择。Vin1 指向 VDDA，VDDA 为 3.3V。Vin2 指向 VREF_OUT，VREF_OUT 输出一个精确的 1.2V。

比较器比较两路模拟电压信号的输入，INP 和 INM 为正向输入通道和反向输入通道。INP 和 INM 比较通道分别从两个模拟 MUX 选择一路输入信号。当正向输入大于反向输入时，模拟信号输出(CMPO)为高，当正向输入小于反向输入时，CMPO 为低。KL25 中输出信号还可以通过 CR1 寄存器中的 INV 置 1 来确定反向输出。若两个输入通道选择了一样的输入，那 CMP 模块将会自动关闭模块。CMP 模块将从用户选择的通道进行比较。若要使用 DAC 输出的比较电压做参考电压，设置通道号为 7 即可。

2）CMP 模块内部结构及工作过程

图 10-7 为比较器内部结构图。由比较器、极性选择(Polarity Select)、窗口控制(Window Control)、滤波器模块(Filter Block)、中断控制(Interrupt Control)和时钟分频器(Clock Prescaler)模块组成。

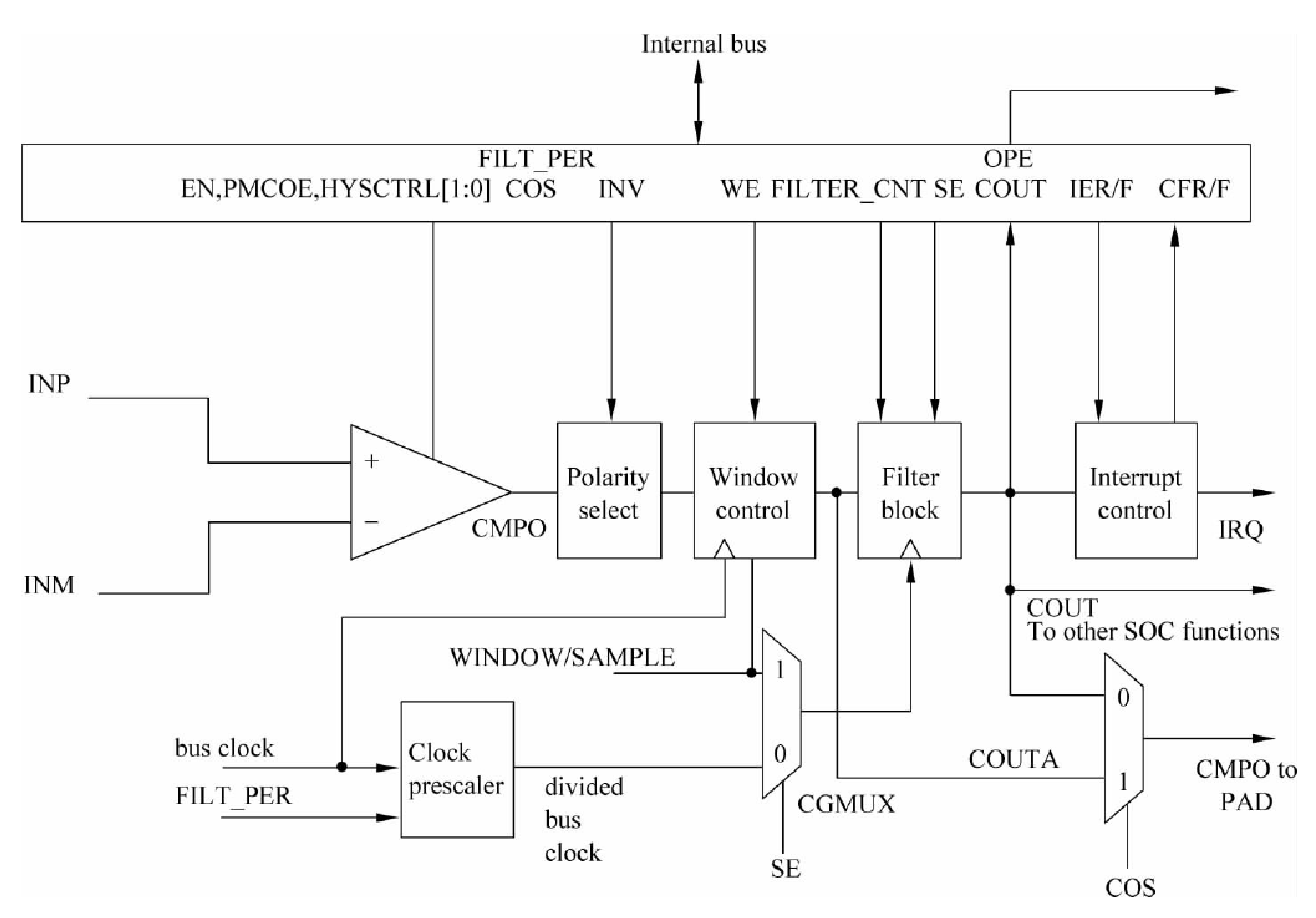


图 10-7　比较器模块结构图

首先，比较器输出信号为 CMPO 通过极性选择模块控制输出的 CMPO 是否反转。其次，CMPO 信号到 Window 控制器，Window 控制器相当于一个门，当 Window＝1 时，输出 COUTA＝CMPO；当 Window＝0 时，COUTA 保持原状，CMPO 无法通过。再次，COUTA 信号通过滤波模块进行采样和滤波后输出 COUT 信号，COUT 信号可以直接从芯片引脚输出，也可以输出到其他模块，根据 COUT 输出波形的上升沿和下降沿可以配置中断使能。

当然，CMPO 信号可以直接越过 Window 控制器，此时 COUTA＝CMPO；也可以不进行采样和滤波，此时 COUT＝COUTA＝CMPO，具体依据控制寄存器的配置，不同的配置形成了不同 CMP 工作模式。

3）CMP 模块工作模式

根据控制寄存器 CR1[EN]（比较器使能位）、CR1[WE]（窗口使能位）、CR1[SE]（采样使能位）、CR0[FILTER_CNT]（滤波次数）、FPR[FILT_PER]（滤波周期数）等位不同组合配置形成不同的 CMP 工作模式。CMP 工作模式如表 10-23 所示。

表 10-23　CMP 工作模式

工作模式	CR1 [EN]	CR1 [WE]	CR1 [SE]	CR0 [FILTER_CNT]	FPR [FILT_PER]	操　　作
禁止模式	0	X	X	X	X	比较器停止工作不耗电。CMPO 输出全是 0
持续模式	1	0	0	0x00	X	窗口控制器和滤波器不可用，可配置翻转。COUT/COUTA/CMPO 是同一个信号
	1	0	0	X	0x00	
采样无滤波模式	1	0	1	0x01	X	COUTA＝CMPO，采样 COUTA 产生 COUT SE＝1 采样周期由外部时钟控制，SE＝0 采样周期由 FPR[FILT_PER]控制
	1	0	0	0x01	>0x00	
采样滤波模式	1	0	1	>0x01	X	COUTA＝CMPO，采样 COUTA 滤波后产生 COUT，SE＝1 采样周期由外部时钟控制，SE＝0 采样周期由 FPR[FILT_PER]控制
	1	0	0	>0x01	>0x00	
窗口模式	1	1	0	0x00	X	当 Window＝1 时，每个总线时钟的上升沿 CMPO 传给 COUTA，并传给 COUT
	1	1	0	X	0x00	
窗口/重复采样模式	1	1	0	0x01	0x01-0xff	当 Window＝1 时，每个总线时钟的上升沿 CMPO 产生 COUTA，由滤波器重复采样 COUTA 产生 COUT
窗口/滤波模式	1	1	0	>0x01	0x01-0xff	当 Sample＝1 时，每个总线时钟的上升沿采样 CMPO 传给 COUTA，由滤波器重复采样 COUTA 并进行滤波产生 COUT

注：其他任何的组合(CR1[EN]，CR1[WE]，CR1[SE]，CR0[]和 FPR[FILT_PER])都是非法的。

滤波器的采样时钟可以选用内部时钟或者外部时钟，CR1[SE]＝1 选用外部时钟控制，CR1[SE]＝0，采样周期由 FPR[FILT_PER]和总线时钟控制。滤波器可以规定滤波采样次数(在 CR0[FILTER_CNT]中)输出比较值。若滤波采样次数为 1，滤波器类似于最简单的采样器，此时没有滤波。

4）CMP 中断

CMP 可以在比较输出的上升沿或者下降沿或者两者都拥有时产生中断。表 10-24 给出了在何种条件下中断请求生效和无效的情况。

表 10-24　CMP 中断条件

条　件	结　果
SCR[IER]和 SCR[CFR]置位	上升沿中断请求生效
SCR[IEF]和 SCR[CFF]置位	下降沿中断请求生效
清除 SCR[IER]和 SCR[CFR]位来取消上升沿中断	中断请求无效
清除 SCR[IEF]和 SCR[CFF]位来取消下降沿中断	中断请求无效

2. CMP 驱动构件源码

1）基本编程方法

(1) 首先计算出比较器模块以及各个寄存器的基址，打开 CMP 模块的时钟。

(2) 根据参数的正向输入和负向输入配置 CMP_MUXCR 寄存器的正向输入（PSEL）位和负向输入（MSEL）位；使能 CMP_DACCR 寄存器的输入（DACEN）位，使能 DAC 输入；根据参考电压参数选择 DAC 参考电压（VRSEL）位（V_{DDA}或 V_{REF}）。

完成上述的寄存器操作后，CMP 模块便开始工作了，下面开始设置 DAC 的输入值。

(3) 设置 DAC 模拟输出的值，配置 CMP_VOSEL 寄存器，输出对应的 DAC 电压。

完成 DAC 的输入后，还需要设置比较器输出结果能够产生中断。

(4) 注册 CMP 中断，中断号为 16。

(5) 使能 CMP。

上述操作完成后，便可以使 CMP 模块工作起来了。

2）CMP 驱动构件源程序文件

```
//========================================================================
//文件名称：cmp.c
//功能概要：KL25 比较器底层驱动程序文件
//版权所有：苏州大学恩智浦嵌入式中心(sumcu.suda.edu.cn)
//版本更新：2012-11-25   V1.0    初始版本
//========================================================================
#include "cmp.h"

//========================================================================
//函数名称：cmp_init
//函数返回：无
//参数说明：reference:参考电压选择   0=Vin1in   1=Vin2in
//          plusChannel：正比较通道号
//          minusChannel：负比较通道号
//通道号 0,1,2,3,4,5 对应引脚 PTC6,PTC7,PTC8,PTC9,PTC30,PTC29；通道 6 Bandgap;通道 7
//DAC 输入
//功能概要：CMP 模块初始化
//========================================================================
void cmp_init(uint_8 reference,uint_8 plusChannel,uint_8 minusChannel)
```

```
{
    //通过获取模块号选择比较器基址
    CMP_MemMapPtr cmpch = CMP0_BASE_PTR;
    if(plusChannel>7)
        plusChannel = 7;
    if(plusChannel<0)
        plusChannel = 0;

    if(minusChannel>7)
        minusChannel = 7;
    if(minusChannel<0)
        minusChannel = 0;

    //使能比较模块时钟
    SIM_SCGC4 |=SIM_SCGC4_CMP_MASK;

    //初始化寄存器
    CMP_CR0_REG(cmpch) = 0;
    CMP_CR1_REG(cmpch) = 0;
    CMP_FPR_REG(cmpch) = 0;
    //如果设置了标志清除中断标志
    CMP_SCR_REG(cmpch) = 0x06;
    CMP_DACCR_REG(cmpch) = 0;
    CMP_MUXCR_REG(cmpch) = 0;

    //配置寄存器
    //过滤,数字延时禁止
    CMP_CR0_REG(cmpch) = 0x00;
    //连续模式,高速比较,无过滤输出,输出引脚禁止
    CMP_CR1_REG(cmpch) = 0x16;
    //过滤禁止
    CMP_FPR_REG(cmpch) = 0x00;
    //使能上升沿和下降沿中断,清标志位
    CMP_SCR_REG(cmpch) = 0x1E;

    if(reference==0)                                      //参考电压选择 VDD3.3V
    {
        //6 位参考 DAC 使能,选择 VDD 作为 DAC 参考电压
        CMP_DACCR_REG(cmpch) |= 0xC0;
    }
    else
    {
        //6 位参考 DAC 使能,选择 VREF 作为 DAC 参考电
        CMP_DACCR_REG(cmpch) |= 0x80;
    }

    CMP_MUXCR_REG(cmpch) = CMP_MUXCR_PSEL(plusChannel)    //正通道选择
                        | CMP_MUXCR_MSEL(minusChannel);   //负通道选择

    //选择输出引脚
```

```
    PORTC_PCR5=PORT_PCR_MUX(6);
    //CMP 使能
    CMP_CR1_REG(cmpch) |= CMP_CR1_EN_MASK;
}

//===========================================
//函数名称: dac_set_value
//函数返回: 无
//参数说明: value: dac 输出的转换值
//功能概要: 设置 DAC 输出值
//===========================================
void dac_set_value(uint_8 value)
{
    //通过获取模块号选择比较器基址
    CMP_MemMapPtr cmpch = CMP0_BASE_PTR;
    CMP_DACCR_REG(cmpch) |= CMP_DACCR_VOSEL(value);
}

//===========================================
//函数名称: cmp_enable_int
//函数返回: 无
//参数说明: 无
//功能概要: 开比较中断
//===========================================
void cmp_enable_int()
{
    //通过获取模块号选择比较器基址
    CMP_MemMapPtr cmpch = CMP0_BASE_PTR;
    //开放 cmp 接收中断,上升沿下降沿均触发
    CMP_SCR_REG(cmpch)|= CMP_SCR_IEF_MASK | CMP_SCR_IER_MASK;
    enable_irq(16);
}

//===========================================
//函数名称: cmp_disable_int
//函数返回: 无
//参数说明: 无
//功能概要: 关比较中断
//===========================================
void cmp_disable_int()
{
    //通过获取模块号选择比较器基址
    CMP_MemMapPtr cmpch = CMP0_BASE_PTR;
    //关闭 cmp 接收中断,上升沿下降沿均关闭
    CMP_SCR_REG(cmpch)&=(~CMP_SCR_IEF_MASK) | (~CMP_SCR_IER_MASK);
    //关接收引脚的 IRQ 中断
    disable_irq(16);

}
```

3）CMP中断函数文件isr.c

```
//==========================================================
//函数名：cmp0_isr
//功能：比较器输出上升沿下降沿中断触发
//说明：无
//==========================================================
//比较器中断处理函数
void CMP0_IRQHandler(void)
{
    //如果是上升沿
    if ((CMP0_SCR & CMP_SCR_CFR_MASK)==CMP_SCR_CFR_MASK)
    {
        CMP0_SCR |= CMP_SCR_CFR_MASK;          //清标志
        uart_send_string(UART_1, "\r\nRising edge on HSCMP0\r\n");
    }
    //如果是下降沿
    if ((CMP0_SCR & CMP_SCR_CFF_MASK) == CMP_SCR_CFF_MASK)
    {
        CMP0_SCR |= CMP_SCR_CFF_MASK;          //清标志
        uart_send_string(UART_1, "\r\nFalling edge on HSCMP0\r\n");
    }
}
```

小　　结

本章在主要阐述了嵌入式系统将模拟量的输入采集转换为数字量以及数字量转换为模拟量，并进行输出比较的处理过程和编程方法。ADC和DAC是嵌入式应用中的重要组成部分，是嵌入式系统与外界连接的纽带，在测控系统中的重要内容，要很好地掌握。

(1) 分析了AD转换过程中的转换精度、转换速度、数据滤波和物理回归等基础问题；介绍了AD转换常用的温度传感器、光敏电阻器、灰度传感器等，以及它们的AD采样的电路原理。同时，也给出了电阻型传感器的实际采样电路。

(2) 介绍了KL25线性逐次逼近型的ADC0模块的基本功能。具有2路差分输入，每路通道可以分别配置为16位、13位、11位和9位采样精度；具有14路外部引脚单端输入模式，每路可以分别配置为16位、12位、10位和8位4种采集精度；以及其他形式的模拟输入通道。以表格的形式给出了各模拟量输入通道的通道号和芯片引脚号，便于编程查阅使用。

(3) 详细介绍了ADC转换模块编程时常用的寄存器；状态控制寄存器SC1A和SC1B可以分别各自控制一路模拟量采样通道，SC1A可以通过软件触发也可以通过硬件触发，SC1B只能通过硬件触发采样。在状态控制寄存器SC2可以设置AD转换结果与预先设定值(存放在比较寄存器CV1和CV2中)，只有满足比较条件的采样结果才存入结果寄存器RA或RB中，转换完成标志COCO才会置1，这样可以排除一些干扰信号被采样进来，也可

以有选择地对阈值范围的数据进行采样，忽略不必要采样的数据。

（4）分析了 ADC 转换模块编程方法和编程步骤，给出了 ADC 驱动构件的封装函数 adc.h 和 adc.c，并在网络光盘上给出了测试实例。

（5）分析了 DAC 模块的特性和结构原理，特别指出 KL25 的 DAC 数据缓冲区只有两个 16 位单元，由数据寄存器 DAT0 和 DAT1 组成。它的缓冲区读指针和缓冲区上限指针只需一位表示，分别由控制寄存器 DAC0_C2 的 D4 和 D0 位表示；给出了 DAC 驱动构件的封装函数 dac.h 和 dac.c，并在网络教学资源上给出了测试实例。

（6）分析了 CMP 模块的结构特点和工作原理，详细分析了 CMP 内部结构和工作过程，连续工作模式、采样无滤波和采样滤波工作模式、窗口模式、窗口采样和窗口采样滤波等工作模式，介绍了 CMP 模块编程使用的寄存器，给出了 CMP 输入输出芯片引脚分配表，便于编程查阅。封装 CMP 底层驱动构件的函数 cmp.h 和 cmp.c，并在网络教学资源上给出了测试实例。

习　题

1. 若 A/D 转换的参考电压为 5V，要能区分 0.05mV 的电压，则采样位数为多少？

2. 简述 KL25 的 A/D 转换模块的主要特性。

3. 若总线时钟频率为 20MHz，当 ADC1CFG 寄存器的 ADICLK 位被设置为 01，ADIV 位被设置为 10 时，A/D 采样的频率为多少？

4. 简述 12 位 DAC 模块工作原理。

5. 编写 DAC 模块程序，分别配置缓冲区操作模式为缓冲区正常模式和缓冲区单次扫描模式，完成三角波发生器功能。

6. 简述 CMP 工作原理，理解 CMP 的 6 种功能模式。

第 11 章　SPI、I2C 与 TSI 模块

本章导读：本章主要阐述串行外设接口 SPI、集成电路互连总线 I2C 和触摸感应输入(TSI)模块的工作原理和编程方法。主要内容有：①SPI 的基本原理及编程模型；②I2C 接口的基本原理及编程模型；③TSI 模块的基本知识及一般编程模型。掌握这些接口的编程模型，将能有效地扩展嵌入式系统的功能。触摸感应接口 TSI 作为一种新型的人机交互手段，已应用于越来越多的嵌入式系统中。

本章参考资料：11.1 节(SPI)参考自《KL 参考手册》的第 37 章，11.2 节(I2C)参考自《KL 参考手册》的第 38 章，11.3 节(TSI)参考自《KL 参考手册》的第 42 章。

11.1　串行外设接口 SPI 模块

11.1.1　串行外设接口 SPI 的通用基础知识

1. SPI 基本概念

串行外设接口(Serial Peripheral Interface，SPI)是原摩托罗拉公司推出的一种同步串行通信接口，用于微处理器和外围扩展芯片之间的串行连接，发展成为一种工业标准。目前，各半导体公司推出了大量带有 SPI 的芯片，如 RAM、EEPROM、AD 转换器、DA 转换器、LED/LCD 显示驱动器、I/O 接口芯片、实时时钟、UART 收发器等，为用户的外围扩展提供了灵活而廉价的选择。SPI 一般使用 4 条线：串行时钟线 SCK、主机输入/从机输出数据线 MISO、主机输出/从机输入数据线 MOSI 和从机选择线 $\overline{SS}$。在阐述 SPI 的特性之前，先来了解 SPI 的相关概念与名称。

1）主机与从机的概念

SPI 系统是典型的“主机-从机”(Master-Slave)系统。一个 SPI 系统，由一个主机和一个或多个从机构成，主机启动一个与从机的同步通信，从而完成数据的交换。提供 SPI 串行时钟的 SPI 设备称为 SPI 主机或主设备(Master)，其他设备则称为 SPI 从机或从设备(Slave)。在 MCU 扩展外设结构中，仍使用主机-从机(Master-Slave)概念，此时 MCU 必须工作于主机方式，外设工作于从机方式。图 11-1 为一个 SPI 的基本连接图，这是一个全双工连接，即收发各用一条线。有的传输方式是单线传输，属于半双工连接，很少使用，本书略过。

2）主出从入引脚 MOSI 与主入从出引脚 MISO

主出从入引脚(Master Out/Slave In，MOSI)是主机输出、从机输入数据线。对于 MCU 被设置为主机方式，主机送往从机的数据从该引脚输出。对于 MCU 被设置为从机方式，来自主机的数据从该引脚输入。

主入从出引脚(Master In/Slave Out，MISO)是主机输入、从机输出数据线。对于 MCU

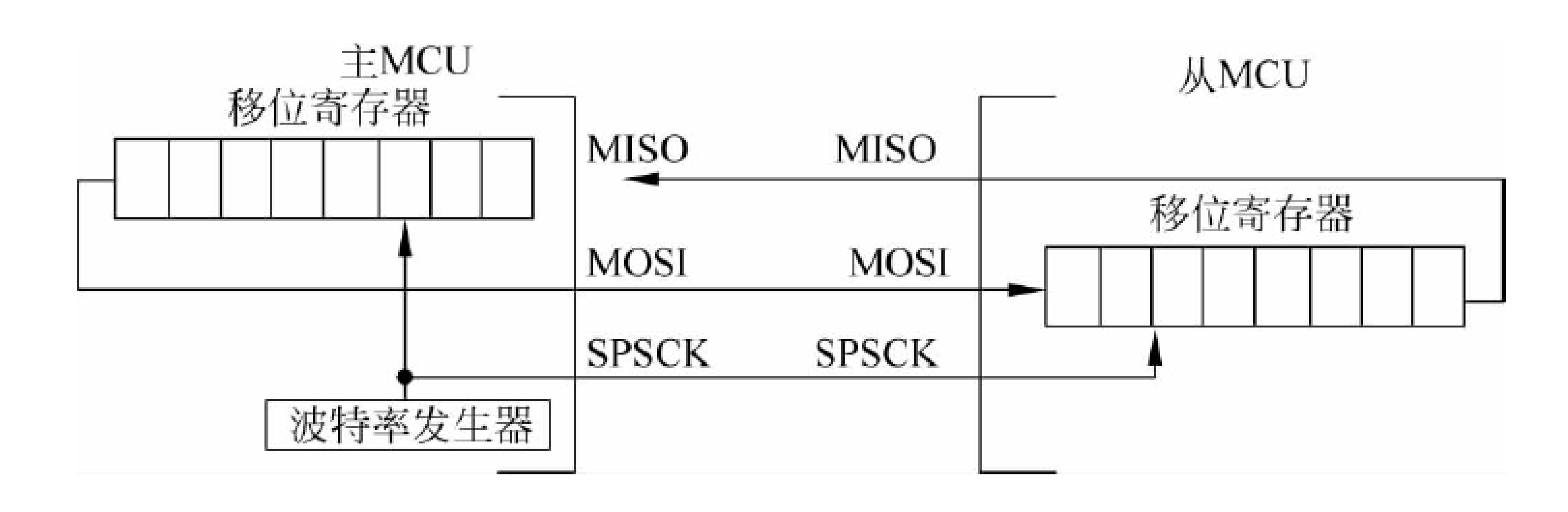


图 11-1　SPI 全双工主-从连接

被设置为主机方式，来自从机的数据从该引脚输入主机。对于 MCU 被设置为从机方式，送往主机的数据从该引脚输出。

3）SPI 串行时钟引脚 SCK

SPI 串行时钟引脚(Serial Clock，SCK)用于控制主机与从机之间的数据传输。串行时钟信号由主机的内部总线时钟分频获得，主机的 SCK 引脚输出给从机的 SCK 引脚，控制整个数据的传输速度。在主机启动一次传送的过程中，从 SCK 引脚输出自动产生的 8 个时钟周期信号，SCK 信号的一个跳变进行一位数据移位传输。

4）时钟极性与时钟相位

时钟极性表示时钟信号在空闲时是高电平还是低电平。时钟相位表示时钟信号 SCK 的第一个边沿出现在第一位数据传输周期的开始位置还是中央位置。

5）从机选择引脚$\overline{SS}$

一些芯片带有从机选择引脚$\overline{SS}$(Slave Select)，也称为片选引脚。若一个 MCU 的 SPI 工作于主机方式，则该 MCU 的$\overline{SS}$引脚设为高电平。若一个 MCU 的 SPI 工作于从机方式，当$\overline{SS}=0$ 时表示主机选中了该从机，反之则未选中该从机。对于单主单从(One Master and One Slave)系统，可以采用图 11-1 的接法。对于一个主 MCU 带多个从属 MCU 的系统，主机 MCU 的$\overline{SS}$接高电平，每一个从机 MCU 的$\overline{SS}$接主机的 I/O 输出线，由主机控制其电平高低，以便主机选中该从机。

2. SPI 的数据传输原理

图 11-1 是 SPI 的主-从连接示意图，图中的移位寄存器为 8 位，所以每一工作过程传送 8 位数据。**从主机 CPU 发出启动传输信号开始，将要传送的数据装入 8 位移位寄存器，并同时产生 8 个时钟信号依次从 SCK 引脚送出，在 SCK 信号的控制下，主机中 8 位移位寄存器中的数据依次从 MOSI 引脚送出，到从机的 MOSI 引脚后送入它的 8 位移位寄存器。在此过程中，从机的数据也可通过 MISO 引脚传送到主机中。**所以，我们称之为全双工主-从连接(Full-Duplex Master-Slave Connections)。其数据的传输格式是高位(MSB)在前，低位(LSB)在后。

图 11-1 是一个主 MCU 和一个从 MCU 的连接，也可以一个主 MCU 与多个从 MCU 进行连接形成一个主机多个从机的系统；还可以多个 MCU 互连构成多主机系统；另外，也可以一个 MCU 挂接多个从属外设。但是，SPI 系统最常见的应用是利用一个 MCU 作为主机，其他处于从机地位，这样，主机的程序启动并控制数据的传送和流向，在主机的控制下，从属设备从主机读取数据或向主机发送数据。至于传送速度、何时数据移入移出、一次移动

完成是否中断和如何定义主机从机等问题,可通过对寄存器编程来解决,下文将阐述这些问题。

3. SPI的时序

SPI的数据传输是在时钟信号SCK(同步信号)的控制下完成的。数据传输过程涉及时钟极性与时钟相位设置问题。以下讲解使用CPOL描述时钟极性,使用CPHA描述时钟相位。主机和从机必须使用同样的时钟极性与时钟相位,才能正常通信。**总体要求是:确保发送数据在一周期开始的时刻上线,接收方在1/2周期的时刻从线上取数,这样是最稳定的通信方式。对发送方编程必须明确两点:接收方要求的时钟空闲电平是高电平还是低电平;接收方在时钟的上升沿取数还是下降沿取数。据此,设置时钟极性与时钟相位。**

关于时钟极性与相位的选择,有4种可能情况,分别如图11-2～图11-5所示。

1)空闲电平低电平,上升沿取数——CPOL＝0,CPHA＝0

若空闲电平为低电平,接收方在时钟的上升沿取数。为了保证数据的正确传输,第一位数据提前半个时钟周期上线,在第一个时钟信号的上升沿,数据已经上线半个周期,处于稳定状态,接收方此时采样线上信号,最为稳定。在时钟信号的一个周期结束后(下降沿),时钟信号又达低电平,下一位数据又开始上线,再重复上述过程,直到一个字节的8位信号传输结束。用CPOL＝0表征空闲电平低电平,用CPHA＝0表征第一位数据提前半个时钟周期上线。则图11-2情况下,CPOL＝0,CPHA＝0。

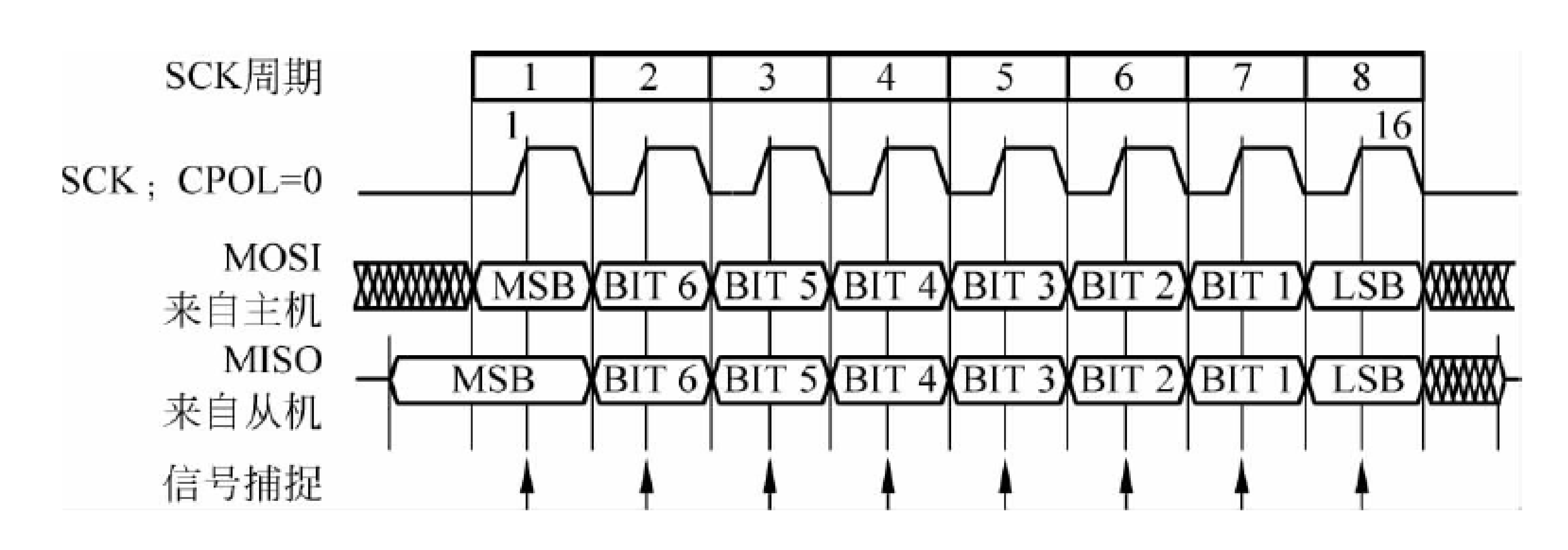


图11-2　CPOL＝0,CPHA＝0时的数据/时钟时序图

2)空闲电平低电平,下降沿取数——CPOL＝0,CPHA＝1

图11-3与图11-2唯一不同之处是在同步时钟信号的下降沿时采样线上信号,类似分析同上。

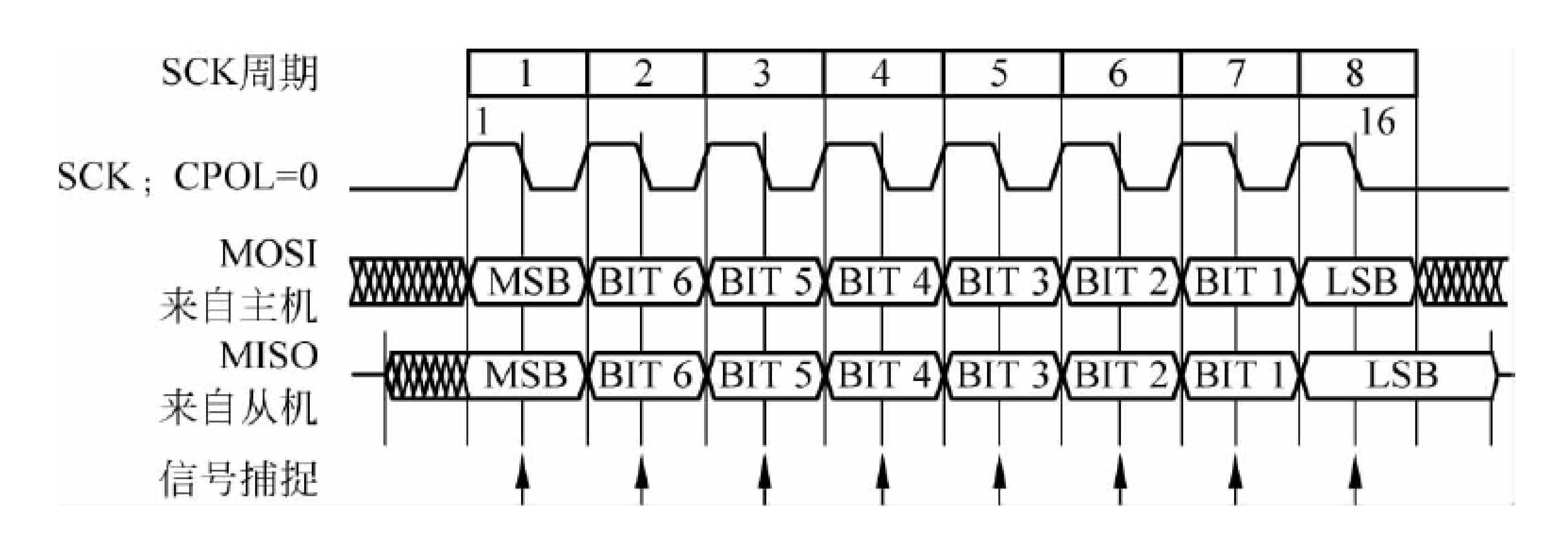


图11-3　CPOL＝0,CPHA＝1时的数据/时钟时序图

3）空闲电平高电平，下降沿取数——CPOL=1，CPHA=0

可对图 11-4 做类似分析。空闲电平高电平（CPOL=1），下降沿取数，数据需提前半个周期上线（CPHA=0）。

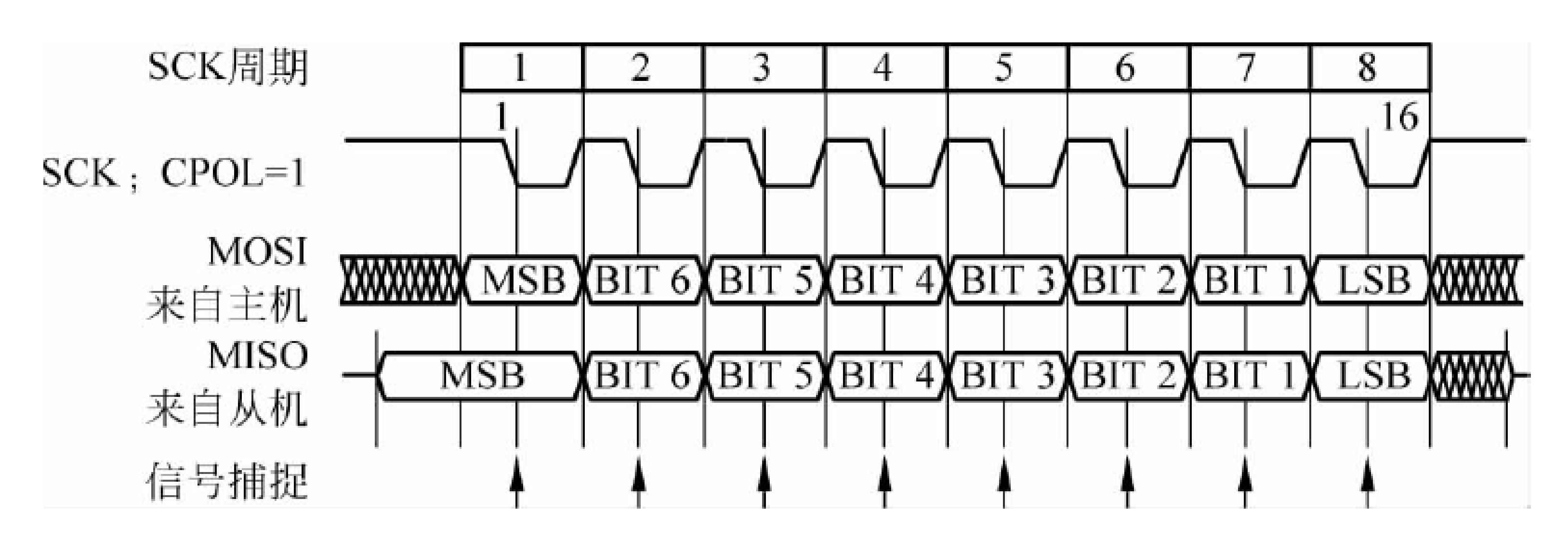


图 11-4 CPOL=1，CPHA=0 时的数据/时钟时序图

4）空闲电平高电平，上升沿取数——CPOL=1，CPHA=0

可对图 11-5 做类似分析。空闲电平高电平（CPOL=1），上升沿取数，数据可与时钟同时变化，不需提前上线（CPHA=1）。

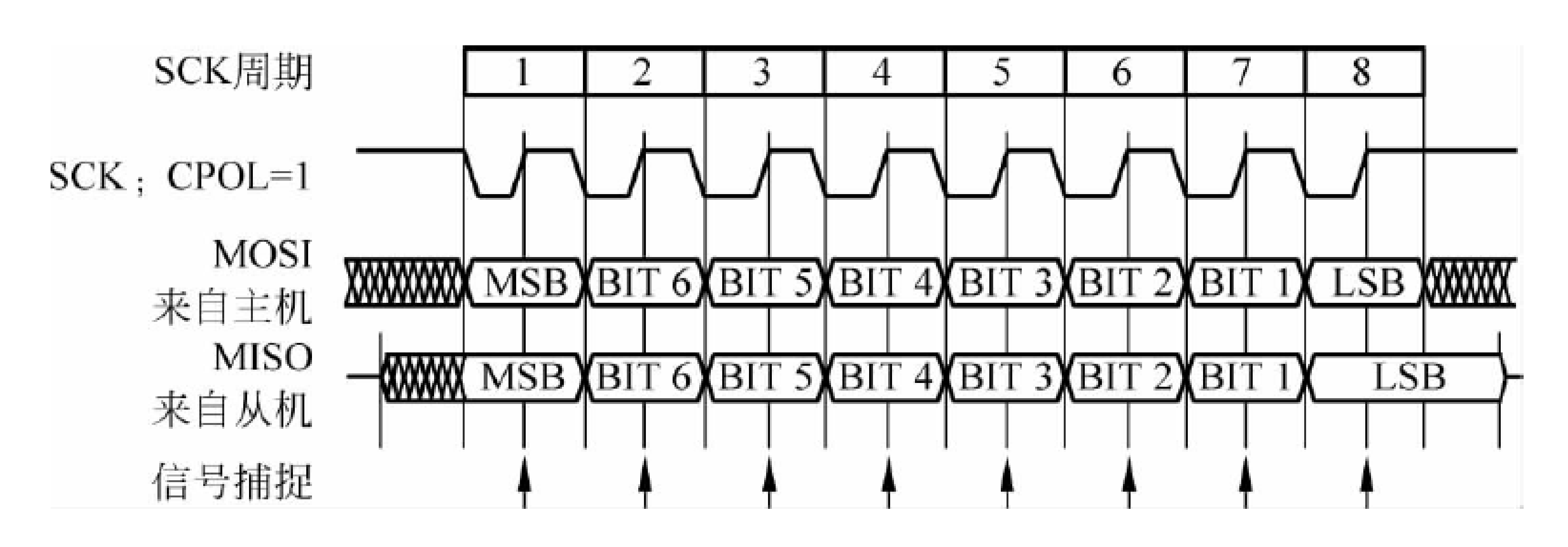


图 11-5 CPOL=1，CPHA=1 时的数据/时钟时序图

只有正确地配置时钟极性和时钟相位，数据才能够被准确地接收。因此必须严格对照从机 SPI 的要求来正确配置主机的时钟极性和时钟相位。

对于不带 SPI 串行总线接口的 MCU 来说，可以使用软件来模拟 SPI 的操作，具体的模拟方法在这里不再详述。本书网上教学资源中的补充阅读材料给出了模拟 SPI 的简要讨论。

11.1.2 SPI 驱动构件头文件及使用方法

1. SPI 引脚

KL25 内部具有两个 SPI 模块，分别是 SPI0 和 SPI1。这两个模块除了时钟源不一样之外，其他的地方完全相同。SPI0 的时钟源是总线时钟，SPI1 的时钟源是系统时钟。在本书用例中总线时钟为 24MHz，系统时钟为 48MHz，因此在设置通信波特率时要注意两者的不同。

表 11-1 给出了 SPI 模块使用的引脚。注意同一 SPI 的 4 根线，可以使用不同引脚组，仅是为了方便硬件布板。实际设计时，从靠近对外接口的最近引脚引出。例如，根据表 11-1，SPI0 可以使用 PTD0～3，也可以使用 PTA14～17，还可以使用 PTC4～7。编程时，使用宏定义确定。

表 11-1　KL25 的 SPI 引脚及 SD-FSL-KL25-EVB 实际使用的引脚

引脚号	引脚名	ALT0	ALT1	ALT2	ALT3	ALT4	ALT5	SD-FSL-KL25-EVB 使用
2	PTE1		PTE1	SPI1_MOSI	UART1_RX		SPI1_MISO	
3	PTE2		PTE2	SPI1_SCK				
4	PTE3		PTE3	SPI1_MOSI			SPI1_MOSI	
5	PTE4		PTE4	SPI1_PCS0				
34	PTA14		PTA14	SPI0_PCS0	UART0_TX			SPI0_PCS0
35	PTA15		PTA15	SPI0_SCK	UART0_RX			SPI0_SCK
36	PTA16		PTA16	SPI0_MOSI		SPI1_MOSI		SPI0_MOSI
37	PTA17		PTA17	SPI0_MISO		SPI1_MISO		SPI0_MISO
49	PTB10		PTB10	SPI1_PCS0				SPI1_PCS0
50	PTB11		PTB11	SPI1_SCK				SPI1_SCK
51	PTB16	TSI0_CH9	PTB16	SPI1_MOSI	UART0_RX	TPM_CLKIN0		SPI1_MOSI
52	PTB17	TSI0_CH10	PTB17	SPI1_MISO	UART0_TX	TPM_CLKIN1		SPI1_MISO
73	PTD0		PTD0	SPI0_PCS0		TPM0_CH0		
74	PTD1	DC0_SE5b	PTD1	SPI0_SCK		TPM0_CH1		
75	PTD2		PTD2	SPI0_MOSI	UART2_RX	TPM0_CH2		
76	PTD3		PTD3	SPI0_MISO	UART2_TX	TPM0_CH3		
77	PTD4		PTD4	SPI1_PCS0	UART2_RX	TPM0_CH4		
78	PTD5	ADC0_SE6b	PTD5	SPI1_SCK	UART2_TX	TPM0_CH5		
79	PTD6	ADC0_SE7b	PTD6	SPI1_MOSI	UART0_RX			
80	PTD7		PTD7	SPI1_MISO	UART0_TX			

2. SPI 驱动构件基本要点分析

SPI 模块具有初始化、发送一个字节、发送 N 个字节、接收一个字节、接收 N 个字节、开中断、关中断等操作。按照构件化的思想，可将它们封装成独立的功能函数。SPI 构件包括头文件 spi.c 和 spi.h 文件。SPI 构件头文件中主要包括相关宏定义、SPI 的功能函数原型说明等内容。SPI 构件程序文件的内容是给出 SPI 各功能函数的实现过程。

在 spi.h 中，给出了用于定义所用 SPI 的宏定义、SPI 所用的引脚组的宏定义。

在 spi.c 中，SPI 的初始化，主要是对 SPI 控制寄存器 SPIx_C1、SPIx_C2 进行设置，定义 SPI 工作模式、时钟的空闲电平及相位、允许 SPI 和对 SPI 的波特率寄存器 SPIx_BR 设置，波特率根据传送速度要求计算而得到。SPI 是一种通信模块，它的基本功能就是接收和发送数据。我们定义了发送单字节的函数 SPI_send1，接收单字节的函数 SPI_receive1。在这两个函数的基础上，又封装了发送多个字节的函数 SPI_sendN，接收多个字节的函数 SPI_receiveN。除此之外还有使能接收中断、关中断函数等。

通过以上分析，可以设计 SPI 构件的几个基本功能函数。

(1) 初始化函数：void SPI_init(uint_8 No, uint_8 MSTR, uint_16 BaudRate, uint_8 CPOL,uint_8 CPHA)；

(2) 发送一字节数据：uint_8 SPI_send1(uint_8 No, uint_8 data)；

(3) 发送 N 字节数据：void SPI_sendN(uint_8 No, uint_8 n, uint_8 data[])；

(4) 接收一字节数据：uint_8 SPI_receive1(uint_8 No)；

（5）接收 *N* 字节数据：uint_8 SPI_receiveN(uint_8 No，uint_8 n，uint_8 data[])；

（6）使能 SPI 中断：void SPI_enable_re_int(uint_8 No)；

（7）关闭 SPI 中断：void SPI_disable_re_int(uint_8 No)；

在 SPI 构件中，含义相同的参数，它们的命名必须是相同的，这样可增加程序的可读性与易维护性。以上 7 个基本功能函数的参数说明如表 11-2 所示。

表 11-2　SPI 基本功能函数参数说明

参　数	含　　义	备　　注
No	模块号	No＝0，表示 SPI0；No＝1，表示 SPI1
MSTR	SPI 主从机选择	MSTR＝0，设为主机；MSTR＝1，设为从机
BaudRate	波特率	可取 12 000、6000、4000、3000、1500、1000，单位：b/s
CPOL	时钟极性	CPOL＝0，空闲电平为低电平；CPOL＝1，空闲电平为高电平
CPHA	时钟相位	当 CPOL＝0，若上升沿取数，则取 CPHA＝0，若下降沿取数，则取 CPHA＝1。当 CPOL＝1，若下降沿取数，则取 CPHA＝0，若上升沿取数，则取 CPHA＝1
n	要发送的字节个数	n 的范围为 1～255
data[]	数组的首地址	

3．SPI 驱动构件头文件

```
//===========================================================
//文件名称：SPI.h
//功能概要：SPI 头文件
//版权所有：苏州大学 NXP 嵌入式中心(sumcu.suda.edu.cn)
//更新记录：2016-03-17      2016-05-11   V3.0
//===========================================================
#ifndef _SPI_H                          //防止重复定义(开头)
#define _SPI_H

#include "common.h"                     //包含公共要素头文件
//宏定义：定义 SPI 口号
#define SPI_0    0                      //SPI0 口
#define SPI_1    1                      //SPI1 口

//根据 SPI 实际硬件引脚，确定以下宏常量值
//在此工程中，只使用 SPI0 组中的第二个，SPI1 组中的第二个，
//因此只需要将 SPI_0_GROUP 宏定义为 2，SPI_1_GROUP 宏定义为 2

//SPI_0：1=PTD0、1、2、3 脚，2=PTA14、15、16、17 脚，3=PTC4、5、6、7 脚
#define SPI_0_GROUP 2                   //SD-FSL-KL25-EVB 板上使用 PTA14～17 脚

//SPI_1：1=PTE1、2、3、4 脚，2=PTB10、11、16、17 脚，3=PTD4、5、6、7 脚
#define SPI_1_GROUP 2                   //SD-FSL-KL25-EVB 板上使用 PTB10、11、16、17 脚


#define SPI_baseadd(SPI_nub) (SPI_MemMapPtr)(0x40076000u+SPI_nub * 0x00001000u)
```

```
//============================================================
//函数名称：SPI_init。
//功能说明：SPI 初始化
//函数参数：No：模块号,KL25 芯片取值为 0、1
//          MSTR：SPI 主从机选择,0 选择为主机,1 选择为从机。
//          BaudRate：波特率,可取 12000、6000、4000、3000、1500、1000,单位：b/s
//          CPOL：CPOL=0：高有效 SPI 时钟(低无效)；CPOL=1：低有效 SPI 时钟(高无效)
//          CPHA：CPHA=0 相位为 0；CPHA=1 相位为 1；
//函数返回：无
//函数备注：CPHA=0,时钟信号的第一个边沿出现在 8 周期数据传输的第一个周期的中央；
//           CPHA=1,时钟信号的第一个边沿出现在 8 周期数据传输的第一个周期的起点。
//           CPHA=0 时,通信最稳定,即接收方在 1/2 周期的时刻从线上取数。
//============================================================
void SPI_init(uint_8 No,uint_8 MSTR,uint_16 BaudRate,uint_8 CPOL,\
                                                  uint_8 CPHA);

//============================================================
//函数名称：SPI_send1.
//功能说明：SPI 发送一字节数据
//函数参数：No：模块号。其取值为 0 或 1
//          data：需要发送的一字节数据
//函数返回：0：发送失败；1：发送成功
//============================================================
uint_8 SPI_send1(uint_8 No,uint_8 data);

//============================================================
//函数名称：SPI_sendN.
//功能说明：SPI 发送数据
//函数参数：No：模块号。其取值为 0 或 1
//          n：要发送的字节个数。范围为 1～255
//          data[]：所发数组的首地址
//函数返回：无
//============================================================
void SPI_sendN(uint_8 No,uint_8 n,uint_8 data[]);

//============================================================
//函数名称：SPI_receive1.
//功能说明：SPI 接收一个字节的数据
//函数参数：No：模块号。其取值为 0 或 1
//函数返回：接收到的数据。
//============================================================
uint_8 SPI_receive1(uint_8 No);

//============================================================
//函数名称：SPI_receiveN.
//功能说明：SPI 接收数据。当 n=1 时,就是接收一个字节的数据…
//函数参数：No：模块号。其取值为 0 或 1
//             n：要发送的字节个数。范围为 1～255
//          data[]：接收到的数据存放的首地址
//函数返回：1：接收成功,其他情况：失败
```

```
//============================================================
uint_8 SPI_receiveN(uint_8 No,uint_8 n,uint_8 data[]);

//============================================================
//函数名称：SPI_enable_re_int
//功能说明：打开 SPI 接收中断
//函数参数：No：模块号。其取值为 0 或 1
//函数返回：无
//============================================================
void SPI_enable_re_int(uint_8 No);

//============================================================
//函数名称：SPI_disable_re_int
//功能说明：关闭 SPI 接收中断
//函数参数：No：模块号。其取值为 0 或 1
//函数返回：无
//============================================================
void SPI_disable_re_int(uint_8 No);

#endif   //防止重复定义(结尾)
```

4. SPI 驱动构件使用方法

下面以 KL25 中 SPI_0 和 SPI_1 之间的通信为例，介绍 SPI 构件的使用方法。

(1) 首先在 SPI 驱动构件头文件(spi.h)中宏定义引脚组。SPI0 使用引脚为 PTA14～17，SPI1 使用引脚为 PTB10、PTB11、PTB16、PTB17 引脚。

(2) 在主函数 main 中，初始化 SPI 模块，具体的参数包括 SPI 所用的端口号、波特率、时钟极性、时钟相位。这里设置的是 SPI0 初始化为主机，SPI1 初始化为从机。

```
//把 SPI0 初始化为主机,波特率 6000,时钟极性 0,时钟相位 0
SPI_init(SPI_0,1,6000,0,0);
//把 SPI1 初始化为从机,波特率 6000,时钟极性 0,时钟相位 0
SPI_init(SPI_1,0,6000,0,0);
```

(3) 开 SPI1 的接收中断。因为 SPI1 被初始化为从机，所以需要开 SPI1 的接收中断，用于接收从主机发送来的数据。

```
SPI_enable_re_int(SPI_1);        //从机 SPI_1 的接收中断
```

(4) 在主循环中，通过 SPI 发送一个字节函数，把一个字节数据通过主机发送出去，然后把数据加 1。

```
SPI_send1(SPI_0,TransferTemp);
TransferTemp++;
```

其中，TransferTemp 为要发送的字节，初始化为字符'A'。

(5) 在中断函数服务例程中，通过 SPI1 接收中断服务程序，接收主机发送过来的一个

字节数据。

```
uint_8 redata;
redata=SPI_receive1(SPI_1);        //接收主机发送过来的一个字节数据
```

然后可以通过串口把接收到的数据发到PC。

5. SPI驱动构件测试实例

为使读者直观地了解SPI模块之间传输数据的过程,SPI驱动构件测试实例使用串口将SPI0和SPI1模块之间传输的数据显示在PC上。测试工程位于网上教学资源中的"..\KL25-program\ch11-KL25-SPI-I2C-TSI"文件夹中,硬件连接见工程文档。测试工程功能如下。

(1) 使用串口1与外界通信,波特率为9600,无校验。

(2) 启动串口接收中断,回发接收数据。

(3) 初始化SPI0和SPI1,SPI0模块作为主机,SPI1模块作为从机。

(4) 启动SPI1接收中断,将SPI1接收到的数据通过串口发送到PC。

(5) 在main.c的主循环中SPI0向SPI1发送字符A~Z,SPI1通过中断接收到这些字符并判断,并通过串口发送给PC。

(6) 复位之后输出"This is a SPI Test!"。

11.1.3 SPI模块的编程结构

KL25的SPI0模块共有6个8位寄存器,包括两个控制寄存器,一个波特率寄存器,一个状态寄存器,一个数据寄存器及一个匹配寄存器。通过对这些寄存器的编程,就可以使用SPI模块进行数据传输。

1. SPI控制寄存器

1) 控制寄存器1——SPIx_C1

SPIx_C1包括SPI使能控制,中断使能和配置选项三个功能,其结构如表11-3所示。SPI控制寄存器一般情况下只能复位时写一次,以后不再对其写入,不再更改对SPI的设置。SPIx_C1复位后各位为0。

表11-3　SPIx_C1结构

数据位	D7	D6	D5	D4	D3	D2	D1	D0
读/写	SPIE	SPE	SPTIE	MSTR	CPOL	CPHA	SSOE	LSBFE

D7——SPIE为SPI中断使能(用于SPRF和MODF)。这是SPI接收缓冲器已满(SPRF)和模式故障(MODF)事件的中断使能。SPIE=0,禁止SPRF和MODF中断;SPIE=1,当SPRF或MODF为1时,请求硬件中断。

D6——SPE为SPI系统使能位,为SPI系统指定SPI端口引脚功能。如果SPE清零,则SPI被禁止并强制进入空闲状态,在S寄存器中的所有状态位将被重置。

D5——SPTIE为SPI发送中断使能,这是SPI发送缓冲器(SPTEF)的中断使能位。当SPI传送缓冲区空时产生中断。SPTIE=0,禁止SPTEF中断;SPTIE=1,请求硬件中断。

D4——MSTR 为主/从模式选择。MSTR=0,SPI 模块配置为从 SPI 器件；MSTR=1,SPI 模块配置为主 SPI 器件。

D3——CPOL 为时钟极性位。这个位有效地对从主 SPI 到从 SPI 器件的时钟信号串联放置一个反相器。CPOL=0,高有效 SPI 时钟(低无效)；CPOL=1,低有效 SPI 时钟(高无效)。

D2——CPHA 为时钟相位位。该位选择两种时钟格式的一种用于不同类型的同步串行外围器件。CPHA=0,SPSCK 上的第一个边沿出现在 8 周期数据传输的第一个周期的中央；CPHA=1,SPSCK 上的第一个边沿出现在 8 周期数据传输的第一个周期的起点。

D1——SSOE 为辅的选择输出使能。该位的使用需要结合 SPIx_C2 中的模式故障使能(MODFEN)位和主从控制位(MSTR),以确定$\overline{SS}$引脚的功能。当 SSOE=0 时,若 MODFEN=0,则在主模式下,$\overline{SS}$引脚功能配置为通用 I/O(非 SPI),在从模式下,$\overline{SS}$引脚功能配置为从选择输入；若 MODFEN=1,则在主模式下,$\overline{SS}$引脚功能配置为模式故障的$\overline{SS}$输入,在从模式下,$\overline{SS}$引脚功能配置为从选择输入；当 SSOE=1 时,若 MODFEN=0,则在主模式下,$\overline{SS}$引脚功能配置为通用 I/O(非 SPI),在从模式下,$\overline{SS}$引脚功能配置为从选择输入；若 MODFEN=1,则在主模式下,$\overline{SS}$引脚功能配置为$\overline{SS}$从机选择输出,在从模式下,$\overline{SS}$引脚功能配置为从选择输入。

D0——LSBFE 为 LSB 优先(移位器方向)。LSBFE=0,SPI 串行数据传输以最高有效位开始；LSBFE=1,SPI 串行数据传输以最低有效位开始。

2) 控制寄存器 2——SPIx_C2

SPIx_C2 寄存器有硬件匹配中断使能,发送 DMA 使能,主模式故障功能使能,双向模式输出使能,接收 DMA 使能以及 SPI 管脚控制等功能,其结构如表 11-4 所示。复位后各位为 0。

表 11-4　SPIx_C2 结构

数据位	D7	D6	D5	D4	D3	D2	D1	D0
读/写	SPMIE	Reserved	TXDMAE	MODFEN	BIDIROE	RXDMAE	SPISWAI	SPC0

D7——SPMIE 硬件匹配中断功能使能。SPME=0,禁止硬件匹配功能,即使 SPIx_M 数据与收到的数据相同(或者说匹配)也不会触发中断。SPME=1,使能硬件匹配中断,当 SPIx_M 值与接收的数据相同时(或者说匹配时),就会触发一个中断。

D6——保留位且总是读 0。

D5——TXDMAE 发送 DMA 使能位,当 TXDMAE=0,DMA 传输请求被禁止,SPTEF 中断允许。当 TXDMAE=1,DMA 传输请求被打开,SPTEF 中断关闭。

D4——MODFEN 为主模式故障功能使能。当为从模式配置 SPI 时,该位没有意义或影响(SS 管脚是从选择输入)。在主模式中,该位决定 SS 管脚的使用方式。MODFEN=0,模式故障功能禁止,主 SS 管脚恢复为不受 SPI 控制的通用 I/O；MODFEN=1,模式故障功能使能,主 SS 管脚用作模式故障输入或辅(从)选择输出。

D3——BIDIROE 为双向模式输出使能——双向模式由 SPI 管脚控制 0(SPC0=1)使能时,BIDIROE 决定 SPI 数据输出驱动器是否被使能为单个双向 SPI 的 I/O 管脚。根据

SPI 是配置为主 SPI 还是从 SPI，它将 MOSI（MOMI）或 MISO（SISO）管脚分别用作单个 SPI 数据 I/O 管脚，当 SPC0＝0，BIDIROE 没有意义或影响。BIDIROE＝0，输出驱动器禁止，因此 SPI 数据 I/O 管脚作为输入；BIDIROE＝1，SPI I/O 管脚作为输出使能。

D2——RXDMAE 接收 DMA 使能位。RXDMAE＝0，接收 DMA 请求被禁止，SPRF 中断允许。RXDMAE＝1，接收 DMA 请求被打开，SPRF 中断关闭。

D1——SPISWAI 为 SPI 停止在等待模式中。SPISWAI＝0，在等待模式中 SPI 时钟继续运行；SPISWAI＝1，当 MCU 进入等待模式时 SPI 时钟停止。

D0——SPC0 为 SPI 管脚控制 0。SPC0 位用于选择单线双向模式。SPC0＝0，使用分开的引脚用于数据输入和数据输出（引脚模式是正常的）；SPC0＝1，SPI 配置为单线双向操作 MOMI 和 SISO。

2. SPI 波特率寄存器 SPIx_BR

该寄存器用于为一个 SPI 主机设定预定标器和位速率分频因子，其结构如表 11-5 所示。它可以在任何时间被读取或写入，复位为 0。

表 11-5　SPIx_BR 结构

数据位	D7	D6～D4	D3～D0
读	0	SPPR[2:0]	SPR[3:0]
写	—		

D7——保留位且总是读 0。

D6～D4——SPPR[2:0]为 SPI 波特率预分频系数。由这三位位段可以得到 SPI 波特率预分频系数，方法如下。设 SPPR[2:0]所表示的十进制数为 x，则预分频系数 SPPR＝$x+1$；例如 SPPR[2:0]为“010”，即 $x=2$，由此可以得到预分频系数为 3。该预分频器的输入是总线速率时钟（BUSCLK），其输出驱动 SPI 波特率系数的输入，如图 11-6 所示。

D3～D0——SPR[3:0]为 SPI 波特率系数。由这 4 位位段可以得到波特率系数，计算方法如下。设 SPR[3:0]所表示的十进制数为 y，则波特率系数 SPR＝$2^{(y+1)}$。例如 SPR[3:0]＝“0100”，那么 $y=4$，则其 SPR＝$2^{(4+1)}=32$。其输入来自 SPI 波特率预分频器，输出是主模式的 SPI 波特率时钟。

SPI 波特率生成示意图如图 11-6 所示，由总线时钟获得主模式 SPI 的波特率的公式如下：

$$\text{SPI 主模式波特率} = f_{\text{BUSCLK}}/(\text{SPPR} \times \text{SPR})$$

其中，f_{BUSCLK}为总线时钟，SPPR 和 SPR 可分别由 SPI 波特率预分频系数和 SPI 波特率系数得出，计算方法在这里不再重复叙述。

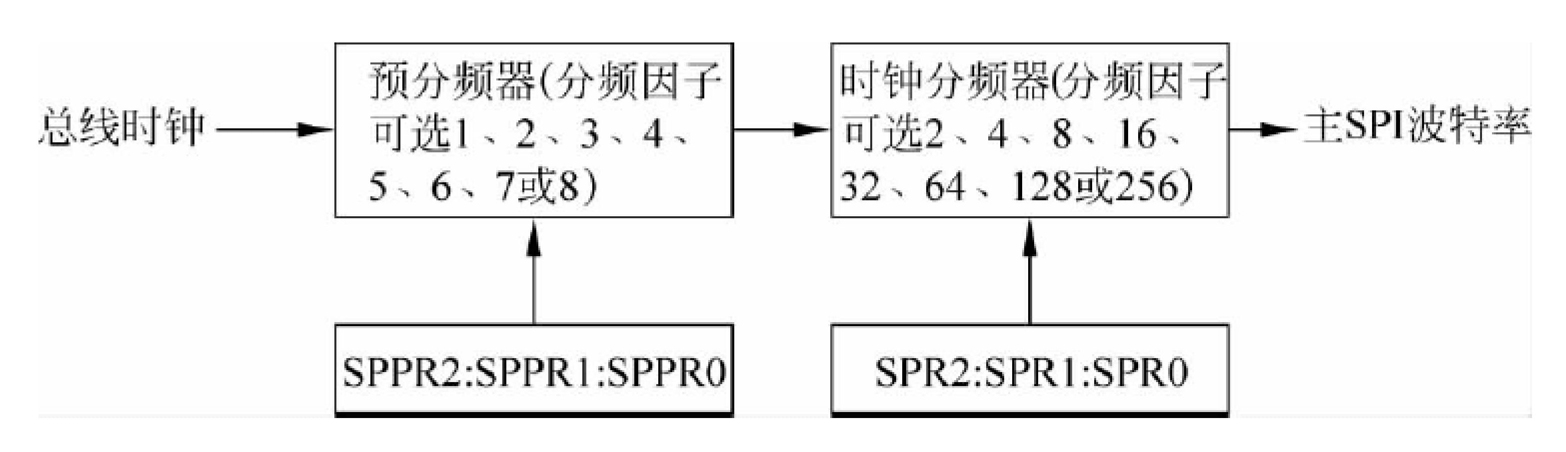


图 11-6　SPI 波特率生成

3. SPI 状态寄存器 SPIx_S

SPIx_S 寄存器主要用于 SPI 发送和接收缓冲区满空的判断和设置，其结构如表 11-6 所示。其中 4 个只读状态位，位 3、2、1 和 0 没有定义，复位为 0，写入无效。

表 11-6　SPIx_S 结构

数据位	D7	D6	D5	D4	D3～D0
读/写	SPRF	SPMF	SPTEF	MODF	0
复位	0	0	1	0	0

D7——SPRF 为 SPI 接收缓冲器满标志。在一次 SPI 传输完成时，SPRF 被置位，表明接收到的数据可以从 SPI 数据寄存器(SPI_D)读取。当 SPRF 被置位时，通过读 SPRF，然后读取 SPI 数据寄存器，可将其清除。SPRF＝0，在接收数据缓冲器中无可用数据；SPRF＝1，在接收数据缓冲器中的数据可用。

D6——SPMF 为 SPI 匹配标志。SPMF＝0 时，在接收数据缓冲区的值不匹配的 M 寄存器中的值。SPMF＝1 时，接收数据缓冲区的值匹配 M 寄存器中的值。要想清除该标志位，先读再写 1。

D5——SPTEF 为 SPI 发送缓冲器空标志。SPTEF＝0，SPI 发送缓冲器非空；SPTEF＝1，SPI 发送缓冲器空。通过读取 SPIx_S，可将其清除，向 SPIx_D 写数据之前，需清 SPTEF 位，否则写到 SPIx_D 的数据无效。在 DMA 方式下，产生 DMA 请求后，SPTEF 自动清除 0。

D4——MODF 为主模式故障标志——如果 SPI 被设置为主模式，且从模式选择输入变为低电平时，MODF 被置位，这表明某个其他的 SPI 器件也被设置成了主模式。只有 MSTR＝1，MODFEN＝1，且 SSOE＝0 时，管脚才作为模式故障错误输入；否则，MODF 将不会被置位。MODF 为 1 时，通过读 MODF，然后写入 SPI 控制寄存器 1(SPIC1)，可将其清空。MODF＝0，无模式故障错误；MODF＝1，检测到模式故障错误。

D3～D0——保留位。

4. SPI 数据寄存器 SPIx_D

读取该寄存器将返回从接收数据缓冲器中读取的数据。写该寄存器将会把数据写入发送数据缓冲器。当 SPI 被配置为主模式时，写入数据到传输数据缓冲器发起一次 SPI 传输。

除非 SPI 发送缓冲器空标志(SPTEF)被置位，数据不应被写入到发送数据缓冲器，表明发送缓冲器内有空间来排列一个新的发送字节。

在 SPRF 置位后且另一传输完成之前的任意时刻，数据都可以从 SPID 中被读取。在一个新的传输完成前，从接收数据缓冲器读出数据失败，将会引起一个接收丢包状态，并且新传输的数据也会丢失。

5. SP 匹配寄存器 SPIx_M

此寄存器存储硬件比较值，用来与 SPI 接收数据缓冲区中的值进行比较。当 SPI 接收数据缓冲区收到的值等于此硬件比较值时，SPI 匹配标志(SPMF)置位。

11.1.4 SPI驱动构件的设计

1. SPI基本编程步骤

实现简单的SPI数据传输主要涉及以下几个寄存器,控制寄存器(SPI0_C1、SPI0_C2)、波特率寄存器(SPI0_BR)、状态寄存器(SPI0_S)、数据寄存器(SPI0_D)。其中,控制寄存器SPI0_C1用于设置SPI中断使能,SPI使能,主/从模式的选择,时钟极性和时钟相位的配置;控制寄存器SPI0_C2用于硬件匹配中断功能使能和主模式故障功能使能的设置;波特率寄存器SPI0_BR用于为一个主机设定预定标器和位速率分频因子;状态寄存器SPI0_S用于判断SPI接收和发送缓冲区是否已满;数据寄存器SPI0_D主要完成数据的传输,读该寄存器返回接收寄存器中的数据,写该寄存器,把数据写入发送寄存器。基本编程步骤(这里以SPI0为例)如下。

(1) 打开SPI模块时钟源,KL25有SPI0和SPI1,所以根据传入的参数来初始化SIM_SCG4_SPI0。

(2) 引脚复用为SPI0功能。选择PTA14的SS功能,选择PTA15的SCK功能,选择PTA16的MOSI功能,选择PTA17的MIOS功能。

(3) 配置控制寄存器SPI0_C1,使能SPI模块(C1[SPE]=1),配置本模式为主机模式(C1[MSTR]=1),从机选择自动模式(C1[SSOE]=1)。

(4) 配置控制寄存器SPI0_C2,模式故障功能使能(C2[MODFEN]=1)。

(5) 配置波特率寄存器SPI0_BR,为主机设定波特率预分频系数和波特率系数。

(6) 若要读取SPI0发送的数据,先判断状态寄存器SPI0_S的SPRF位是否为空,若为空则接收缓冲区没有数据,否则接收缓冲区收到数据,直接读取数据寄存器SPI0_D中的数据即可。

2. SPI驱动构件源码

```
//===========================================================================
//文件名称：SPI.c
//功能概要：SPI底层驱动构件源文件
//版权所有：苏州大学恩智浦嵌入式中心(sumcu.suda.edu.cn)
//更新记录：2016-03-17   V2.2
//          2016-05-11   V3.0
//===========================================================================
#include "spi.h"
//===========================================================================
//函数名称：SPI_init。
//功能说明：SPI初始化
//函数参数：No：模块号,KL25芯片取值为0、1
//          MSTR：SPI主从机选择,0选择为从机,1选择为主机。
//          BaudRate：波特率,可取12000、6000、4000、3000、1500、1000,单位：b/s
//          CPOL：CPOL=0：高有效SPI时钟(低无效)；CPOL=1：低有效SPI时钟(高无效)
//          CPHA：CPHA=0相位为0；CPHA=1相位为1；
//函数返回：无
//函数备注：CPHA=0,时钟信号的第一个边沿出现在8周期数据传输的第一个周期的中央；
//            CPHA=1,时钟信号的第一个边沿出现在8周期数据传输的第一个周期的起点。
```

```
//        CPHA=0 时,通信最稳定,即接收方在 1/2 周期的时刻从线上取数。
//===========================================================
void SPI_init(uint_8 No,uint_8 MSTR,uint_16 BaudRate,\
                                        uint_8 CPOL,uint_8 CPHA)
{
    uint_8 BaudRate_Mode;
    uint_8 BaudRate_High;
    uint_8 BaudRate_Low;
    if(No<0||No>1)                          //如果 SPI 号参数错误则强制选择 0 号模块
        No=0;
    if(No==0)                                    //初始化 SPI0 功能
    {
        BSET(SIM_SCGC4_SPI0_SHIFT,SIM_SCGC4);    //打开 SPI0 模块时钟
        //引脚复用为 SPI0 功能
        #if (SPI_0_GROUP == 1)
        PORTD_PCR0=(0|PORT_PCR_MUX(0x02));       //选择 PTD0 的 SS 功能
        PORTD_PCR1=(0|PORT_PCR_MUX(0x02));       //选择 PTD1 的 SCK 功能
        PORTD_PCR2=(0|PORT_PCR_MUX(0x02));       //选择 PTD2 的 MOSI 功能
        PORTD_PCR3=(0|PORT_PCR_MUX(0x02));       //选择 PTD3 的 MIOS 功能
        #endif

        #if (SPI_0_GROUP == 2)
        PORTA_PCR14=(0|PORT_PCR_MUX(0x02));      //选择 PTA14 的 SS 功能
        PORTA_PCR15=(0|PORT_PCR_MUX(0x02));      //选择 PTA15 的 SCK 功能
        PORTA_PCR16=(0|PORT_PCR_MUX(0x02));      //选择 PTA16 的 MOSI 功能
        PORTA_PCR17=(0|PORT_PCR_MUX(0x02));      //选择 PTA17 的 MIOS 功能
        #endif

        #if (SPI_0_GROUP == 3)
        PORTC_PCR4=(0|PORT_PCR_MUX(0x02));       //选择 PTC4 的 SS 功能
        PORTC_PCR5=(0|PORT_PCR_MUX(0x02));       //选择 PTC5 的 SCK 功能
        PORTC_PCR6=(0|PORT_PCR_MUX(0x02));       //选择 PTC6 的 MOSI 功能
        PORTC_PCR7=(0|PORT_PCR_MUX(0x02));       //选择 PTC7 的 MIOS 功能
        #endif

        SPI0_C1=0x00;                            //SPI 控制寄存器 1 清零
        BSET(SPI_C1_SPE_SHIFT,SPI0_C1);          //使能 SPI 模块

        //MSTR=1 为主机模式;
        //MSTR=0 为从机模式(因 MSTR 初始值为 0,无须更改)
        (MSTR==1)?BSET(SPI_C1_MSTR_SHIFT,SPI0_C1):\
                            BSET(SPI_C1_SPIE_SHIFT,SPI0_C1);

        //时钟极性配置,CPOL=0,平时时钟为高电平,反之 CPOL=1,平时时钟为低电平
        (0==CPOL)?BCLR(SPI_C1_CPOL_SHIFT,SPI0_C1):\
                            BSET(SPI_C1_CPOL_SHIFT,SPI0_C1);

        //CPHA=0 时钟信号的第一个边沿出现在 8 周期数据传输的第一个周期的中央;
        //CPHA=1 时钟信号的第一个边沿出现在 8 周期数据传输的第一个周期的起点。
        (0 == CPHA)?BCLR(SPI_C1_CPHA_SHIFT,SPI0_C1):\
```

```
                            BSET(SPI_C1_CPHA_SHIFT,SPI0_C1);

        BSET(SPI_C1_SSOE_SHIFT,SPI0_C1);   //SSOE 为 1,MODFEN 为 1,配置本模块为自
                                           //动 SS 输出
        //对 SPI0 的 C1 寄存器配置为主机模式、从机选择自动模式并使能 SPI0 模块
        SPI0_C2=0x00;
        if(MSTR == 1)                      //主机模式
            BSET(SPI_C2_MODFEN_SHIFT,SPI0_C2);
        SPI0_BR = 0x00U;                   //波特率寄存器清零
        //重新设置波特率
        BaudRate_High=0;
        BaudRate_Low=0;
        BaudRate_Mode=12000/BaudRate;
        while(BaudRate_Mode % 2 == 0)
        {
            BaudRate_Mode=BaudRate_Mode/2;
            BaudRate_Low++;
        }
        BaudRate_High=--BaudRate_Mode;
        SPI0_BR=BaudRate_High<<4;           //数值赋给 SPI0_BR 的 SPPR 的 D6D5D4 位
        SPI0_BR|=BaudRate_Low;              //赋值给 SPI0_BR 的 SPR 的 D2D1D0 位

    }
    else                                    //初始化 SPI1 功能
    {
        BSET(SIM_SCGC4_SPI1_SHIFT,SIM_SCGC4);          //打开 SPI1 模块时钟
        //引脚复用为 SPI1 功能
        #if (SPI_1_GROUP == 1)
        PORTE_PCR1=(0|PORT_PCR_MUX(0x02));             //选择 PTE1 的 MOSI 功能
        PORTE_PCR2=(0|PORT_PCR_MUX(0x02));             //选择 PTE2 的 SCK 功能
        PORTE_PCR3=(0|PORT_PCR_MUX(0x02));             //选择 PTE3 的 MIOS 功能
        PORTE_PCR4=(0|PORT_PCR_MUX(0x02));             //选择 PTE4 的 SS 功能
        #endif

        #if (SPI_1_GROUP == 2)
        PORTB_PCR10=(0|PORT_PCR_MUX(0x02));            //选择 PTB10 的 SS 功能
        PORTB_PCR11=(0|PORT_PCR_MUX(0x02));            //选择 PTB11 的 SCK 功能
        PORTB_PCR16=(0|PORT_PCR_MUX(0x02));            //选择 PTB16 的 MOSI 功能
        PORTB_PCR17=(0|PORT_PCR_MUX(0x02));            //选择 PTB17 的 MIOS 功能
        #endif

        #if (SPI_1_GROUP == 3)
        PORTD_PCR4=(0|PORT_PCR_MUX(0x02));             //选择 PTD4 的 SS 功能
        PORTD_PCR5=(0|PORT_PCR_MUX(0x02));             //选择 PTD5 的 SCK 功能
        PORTD_PCR6=(0|PORT_PCR_MUX(0x02));             //选择 PTD6 的 MOSI 功能
        PORTD_PCR7=(0|PORT_PCR_MUX(0x02));             //选择 PTD7 的 MIOS 功能
        #endif

        SPI1_C1=0x00;                                  //SPI 控制寄存器 1 清零
        BSET(SPI_C1_SPE_SHIFT,SPI1_C1);                //使能 SPI 模块
```

```
        //MSTR=1 为主机模式;
        //MSTR=0 为从机模式(因 MSTR 初始值为 0,无须更改)
        (MSTR==1)?BSET(SPI_C1_MSTR_SHIFT,SPI1_C1):\
                                  BSET(SPI_C1_SPIE_SHIFT,SPI1_C1);

        //时钟极性配置,CPOL=0,平时时钟为高电平,反之 CPOL=1,平时时钟为低电平
        (0 == CPOL)?BCLR(SPI_C1_CPOL_SHIFT,SPI1_C1):\
                                  BSET(SPI_C1_CPOL_SHIFT,SPI1_C1);

        //CPHA=0 时钟信号的第一个边沿出现在 8 周期数据传输的第一个周期的中央;
        //CPHA=1 时钟信号的第一个边沿出现在 8 周期数据传输的第一个周期的起点。
        (0 == CPHA)?BCLR(SPI_C1_CPHA_SHIFT,SPI1_C1): \
                                  BSET(SPI_C1_CPHA_SHIFT,SPI1_C1);

        BSET(SPI_C1_SSOE_SHIFT,SPI1_C1);  //SSOE 为 1,MODFEN 为 1,配置本模块为自
                                          //动 SS 输出
        //对 SPI0 的 C1 寄存器配置为主机模式、从机选择自动模式并使能 SPI0 模块。

        BSET(SPI_C1_SPIE_SHIFT,SPI1_C1);  //开本模块的 SPI 中断
        SPI1_C2 = 0x00U;
        if(MSTR == 1)                     //主机模式
            BSET(SPI_C2_MODFEN_SHIFT,SPI1_C2);
        SPI1_BR = 0x00U;
        //重新设置波特率
        BaudRate_High=0;
        BaudRate_Low=0;
        BaudRate_Mode=12000/BaudRate;     //取除数用于寄存器中数据计算
        while(BaudRate_Mode % 2 == 0)
        {
            BaudRate_Mode=BaudRate_Mode/2;
            BaudRate_Low++;
        }
        BaudRate_High=--BaudRate_Mode;
        SPI0_BR=BaudRate_High<<4;         //数值赋给 SPI0_BR 的 SPPR 的 D6D5D4 位
        SPI0_BR|=BaudRate_Low;            //赋值给 SPI0_BR 的 SPR 的 D2D1D0 位
    }
}

//=========================================================================
//函数名称: SPI_send1.
//功能说明: SPI 发送一字节数据
//函数参数: No: 模块号。其取值为 0 或 1
//          data: 需要发送的一字节数据
//函数返回: 0: 发送失败; 1: 发送成功
//=========================================================================
uint_8 SPI_send1(uint_8 No,uint_8 data)
{
    uint_32 i;
    SPI_MemMapPtr baseadd=SPI_baseadd(No);
    while(!(SPI_S_REG(baseadd)&SPI_S_SPTEF_MASK));  //等待发送缓冲区空闲
```

```
        SPI_D_REG(baseadd)=data;            //数据寄存器接收数据
    for(i=0;i<0xFFF0;i++)                   //在一定时间内发送不能完成,则认为发送失败
    {
        if(SPI_S_REG(baseadd)&SPI_S_SPTEF_MASK)        //判断发送缓冲区是否接到数据
        {
            return(1);
        }
    }
    return(0);
}

//==========================================================
//函数名称:SPI_sendN.
//功能说明:SPI 发送数据
//函数参数:No:模块号。其取值为 0 或 1
//         n:要发送的字节个数。范围为 1~255
//         data[]:所发数组的首地址
//函数返回:无
//==========================================================
void SPI_sendN(uint_8 No,uint_8 n,uint_8 data[])
{
    SPI_MemMapPtr baseadd=SPI_baseadd(No);
    uint_32 k;
    for(k=0;k<n;k++)
    {
        //状态寄存器的 SPTEF 位不空
        while(!(SPI_S_REG(baseadd)&SPI_S_SPTEF_MASK));
        SPI_D_REG(baseadd)=data[k];
           SPI_S_REG(baseadd) !=SPI_S_SPTEF_MASK;     //清除 SPTEF 位
    }
}

//==========================================================
//函数名称:SPI_receive1.
//功能说明:SPI 接收一个字节的数据
//函数参数:No:模块号。其取值为 0 或 1
//函数返回:接收到的数据
//==========================================================
uint_8 SPI_receive1(uint_8 No)
{
    SPI_MemMapPtr baseadd=SPI_baseadd(No);
    while(!(SPI_S_REG(baseadd)& SPI_S_SPRF_MASK));      //检测 SPI 是否收到了数据
        return SPI_D_REG(baseadd);
}

//==========================================================
//函数名称:SPI_receiveN.
//功能说明:SPI 接收数据。当 n=1 时,就是接收一个字节的数据……
```

```
//函数参数：No：模块号。其取值为 0 或 1
//           n：要发送的字节个数。范围为 1～255
//        data[]：接收到的数据存放的首地址。
//函数返回：1：接收成功,其他情况：失败。
//========================================
uint_8 SPI_receiveN(uint_8 No,uint_8 n,uint_8 data[])
{
    SPI_MemMapPtr baseadd=SPI_baseadd(No);
    uint_32 m=0;
    while(m<n)
    {
        if(SPI_S_REG(baseadd)&SPI_S_SPRF_MASK)
        {
            data[m]=SPI_D_REG(baseadd);
                m++;
        }
    }
    return(1);
}

//========================================
//函数名称：SPI_enable_re_int
//功能说明：打开 SPI 接收中断
//函数参数：No：模块号。其取值为 0 或 1
//函数返回：无
//========================================
void SPI_enable_re_int(uint_8 No)
{
    enable_irq (No+10);
}

//========================================
//函数名称：SPI_disable_re_int
//功能说明：关闭 SPI 接收中断
//函数参数：No：模块号。其取值为 0 或 1
//函数返回：无
//========================================
void SPI_disable_re_int(uint_8 No)
{
    disable_irq (No+10);
}
```

SPI 除了以上给出的功能示例外，还有 DMA 传输模式、单线双向模式等，这些硬件可选功能，读者可以根据使用需要，自行配置。就 SPI 的通信方面来说，硬件和底层驱动只能提供最基本的功能，然而，要想真正实现两个 SPI 对象之间的流畅通信，还需设计基于 SPI 的高层通信协议。

11.2　集成电路互连总线I2C模块

11.2.1　集成电路互连总线I2C的通用基础知识

I2C(Inter-Integrated Circuit),可翻译为“集成电路互连总线”,有的文献中将其缩写为I^2C或IIC,本书一律使用I2C。主要用于同一电路板内各集成电路模块之间的连接。I2C采用双向2线制串行数据传输方式,支持所有IC制造工艺,简化IC间的通信连接。I2C是PHILIPS公司于20世纪80年代初提出,其后PHILIPS和其他厂商提供了种类丰富的I2C兼容芯片。目前,I2C总线标准已经成为世界性的工业标准。

1. I2C总线的历史概况与特点

1992年,PHILIPS首次发布I2C总线规范Version 1.0,并取得专利。

1998年,PHILIPS发布I2C总线规范Version 2.0,至此标准模式和快速模式的I2C总线已经获得了广泛应用,标准模式传输速率为100kb/s,快速模式下为400kb/s。同时,I2C总线也由7位寻址发展到10位寻址,满足了更大寻址空间的需求。

随着数据传输速率和应用功能的迅速增加,2001年PHILIPS公司又发布了I2C总线规范Version 2.1,完善和扩展了I2C总线的功能,并提出了传输速率可达3.4Mb/s的高速模式,这使得I2C总线能够支持现有及将来的高速串行传输,如EEPROM和Flash存储器等。

目前I2C总线已经被大多数的芯片厂家所采用,较为著名的有ST Microelectronics、Texas Instruments、Xicor、Intel、Maxim、Atmel、Analog Devices和Infineon Technologies等,I2C总线标准已经属于世界性的工业标准。I2C总线始终和先进技术保持同步,但仍然保持向下兼容。

I2C总线在硬件结构上,采用数据和时钟两根线来完成数据的传输及外围器件的扩展,数据和时钟都是开漏的,通过一个上拉电阻接到正电源,因此在不需要的时候仍保持高电平。任何具有I2C总线接口的外围器件,不论其功能差别有多大,都具有相同的电气接口,因此都可以挂接在总线上,甚至可在总线工作状态下撤除或挂上,使其连接方式变得十分简单。对各器件的寻址是软寻址方式,因此结点上没有必需的片选线,器件地址给定完全取决于器件类型与单元结构,这也简化了I2C系统的硬件连接。另外,I2C总线能在总线竞争过程中进行总线控制权的仲裁和时钟同步,不会造成数据丢失,因此由I2C总线连接的多机系统可以是一个多主机系统。

I2C主要特点如下。

(1) 在硬件上,二线制的I2C串行总线使得各IC只需最简单的连接,而且总线接口都集成在IC中,不需另加总线接口电路。电路的简化省去了电路板上的大量走线,减少了电路板的面积,提高了可靠性,降低了成本。在I2C总线上,各IC除了个别中断引线外,相互之间没有其他连线,用户常用的IC基本上与系统电路无关,故极易形成用户自己的标准化、模块化设计。

(2) I2C总线还支持多主控(Multi-mastering),如果两个或更多主机同时初始化数据传输,可以通过冲突检测和仲裁防止数据被破坏。其中,任何能够进行发送和接收的设备都可

以成为主机。一个主机能够控制信号的传输和时钟频率。当然在任何时间点上只能有一个主机。

(3) 串行的 8 位双向数据传输位速率在标准模式下可达 100kb/s，快速模式下可达 400kb/s，高速模式下可达 3.4Mb/s。

(4) 连接到相同总线的 IC 数量只受到总线最大电容(400pF)的限制。但如果在总线中加上 82B715 总线远程驱动器可以把总线电容限制扩展 10 倍，传输距离可增加到 15m。

2. I2C 总线硬件相关术语与典型硬件电路

在讲解 I2C 总线过程中涉及以下术语。

(1) 主机(主控器)：在 I2C 总线中，提供时钟信号，对总线时序进行控制的器件。主机负责总线上各个设备信息的传输控制，检测并协调数据的发送和接收。主机对整个数据传输具有绝对的控制权，其他设备只对主机发送的控制信息做出响应。如果在 I2C 系统中只有一个 MCU，那么通常由 MCU 担任主机。

(2) 从机(被控器)：在 I2C 系统中，除主机外的其他设备均为从机。主机通过从机地址访问从机，对应的从机做出响应，与主机通信。从机之间无法通信，任何数据传输都必须通过主机进行。

(3) 地址：每个 I2C 器件都有自己的地址，以供自身在从机模式下使用。在标准的 I2C 中，从机地址被定义成 7 位(扩展 I2C 允许 10 位地址)。地址 0000000 一般用于发出总线广播。

(4) 发送器与接收器：发送数据到总线的器件被称为发送器，从总线接收数据的器件被称为接收器。

(5) SDA 与 SCL(Serial CLock)：串行数据线(Serial DAta，SDA)，串行时钟线(Serial CLock，SCL)。

图 11-7 给出了一个由 MCU 作为主机，通过 I2C 总线带三个从机的单主机 I2C 总线硬件系统。这是最常用、最典型的 I2C 总线连接方式。注意连接时需要共地。

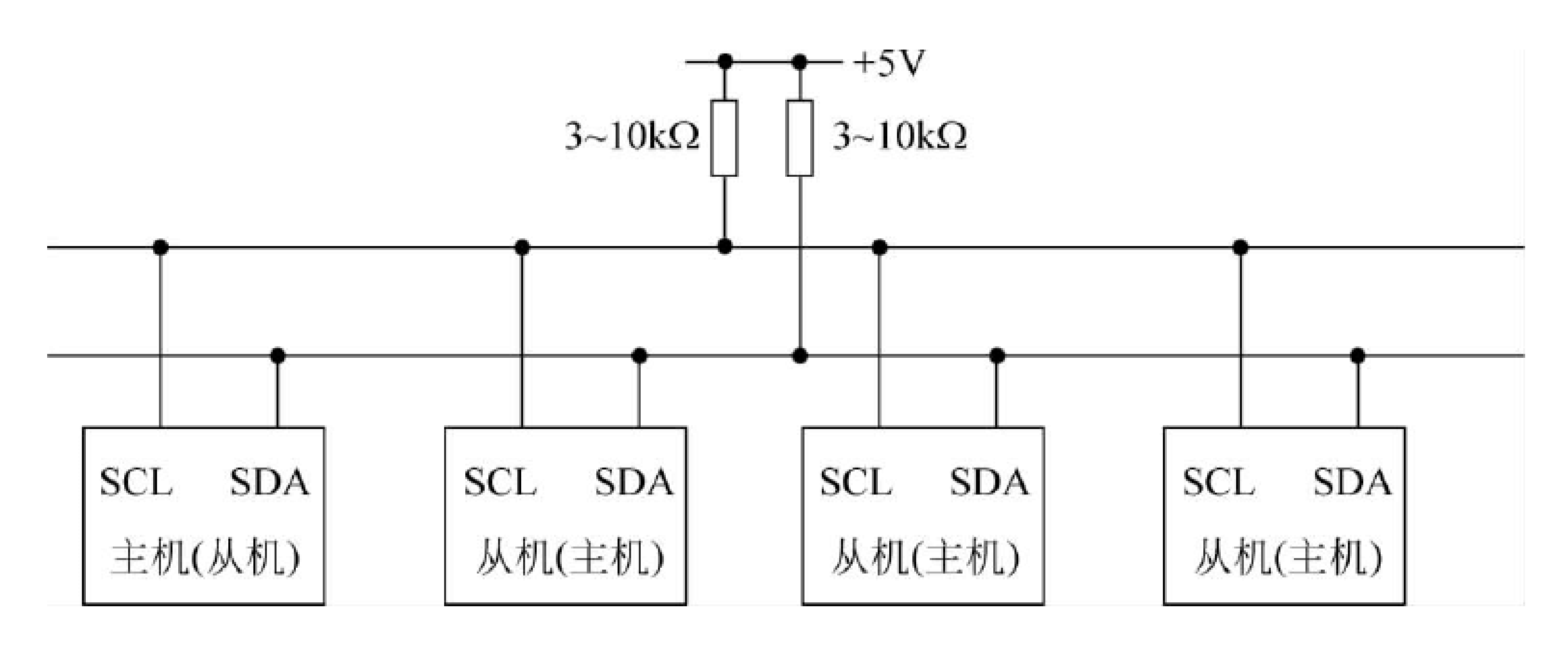


图 11-7　I2C 的典型连接

在物理结构上，I2C 系统由一条串行数据线 SDA 和一条串行时钟线 SCL 组成。SDA 和 SCL 管脚都是漏极开路输出结构，因此在实际使用时，SDA 和 SCL 信号线都必须要加上拉电阻 Rp(Pull-Up Resistor)。上拉电阻一般取值 3～10kΩ，接 5V 电源即可与 5V 逻辑器件接口。主机按一定的通信协议向从机寻址并进行信息传输。在数据传输时，由主机初始化一次数据传输，主机使数据在 SDA 线上传输的同时还通过 SCL 线传输时钟。信息传输的对象和方向以及信息传输的开始和终止均由主机决定。

每个器件都有唯一的地址，且可以是单接收的器件(例如LCD驱动器)，或者是可以接收也可以发送的器件(例如存储器)。发送器或接收器可在主或从机模式下操作。

3. I2C总线数据通信协议概要

1) I2C总线上数据的有效性

I2C总线以串行方式传输数据，从数据字节的最高位开始传送，每个数据位在SCL上都有一个时钟脉冲相对应。在一个时钟周期内，当时钟线高电平时，数据线上必须保持稳定的逻辑电平状态，高电平为数据1，低电平为数据0。当时钟信号为低电平时，才允许数据线上的电平状态变化，如图11-8所示。

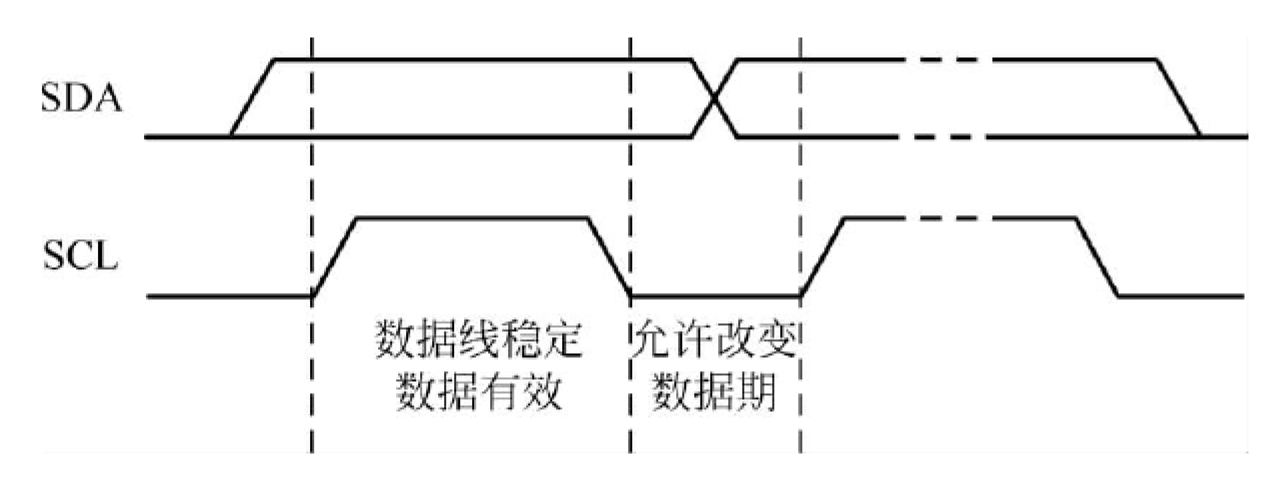


图11-8　I2C总线上数据的有效性

2) I2C总线上的信号类型

I2C总线在传送数据过程中共有4种类型信号，分别是开始信号、停止信号、重新开始信号和应答信号。

开始信号(START)：如图11-9所示，当SCL为高电平时，SDA由高电平向低电平跳变，产生开始信号。当总线空闲的时候(例如，没有主动设备在使用总线，即SDA和SCL都处于高电平)，主机通过发送开始信号(START)建立通信。

停止信号(STOP)：如图11-9所示，当SCL为高电平时，SDA由低电平向高电平跳变，产生停止信号。主机通过发送停止信号，结束时钟信号和数据通信。SDA和SCL都将被复位为高电平状态。

重新开始信号(Repeated START)：在I2C总线上，主机可以在调用一个没有产生STOP信号的命令后，产生一个开始信号。主机通过使用一个重复开始信号来和另一个从机通信或者同一个从机的不同模式通信。由主机发送一个开始信号启动一次通信后，在首次发送停止信号之前，主机通过发送重新开始信号，可以转换与当前从机的通信模式，或是切换到与另一个从机通信。如图11-9所示，当SCL为高电平时，SDA由高电平向低电平跳变，产生重新开始信号，它的本质就是一个开始信号。

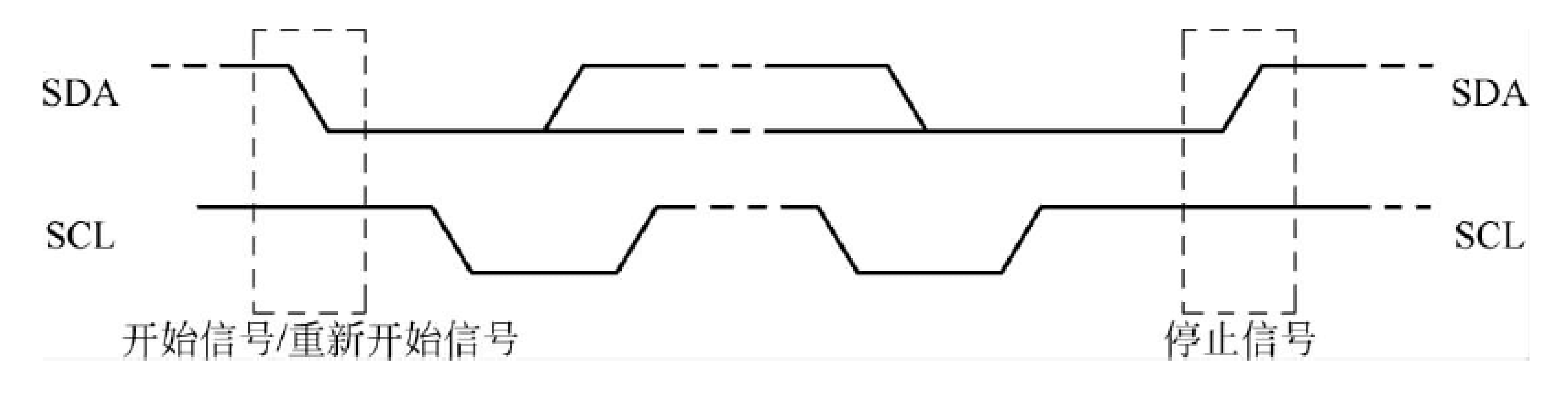


图11-9　开始、重新开始和停止信号

应答信号(A)：接收数据的IC在接收到8位数据后，向发送数据的主机IC发出的特定的低电平脉冲。每一个数据字节后面都要跟一位应答信号，表示已收到数据。应答信号是

在发送了 8 个数据位后，第 9 个时钟周期出现，这时发送器必须在这一时钟位上释放数据线，由接收设备拉低 SDA 电平来产生应答信号，或者由接收设备保持 SDA 的高电平来产生非应答信号，如图 11-10 所示。所以一个完整的字节数据传输需要 9 个时钟脉冲。如果从机作为接收方向主机发送非应答信号，这样主机方就认为此次数据传输失败；如果是主机作为接收方，在从机发送器发送完一个字节数据后，发送了非应答信号表示数据传输结束，并释放 SDA 线。不论是以上哪种情况都会终止数据传输，这时主机或是产生停止信号释放总线，或是产生重新开始信号，从而开始一次新的通信。

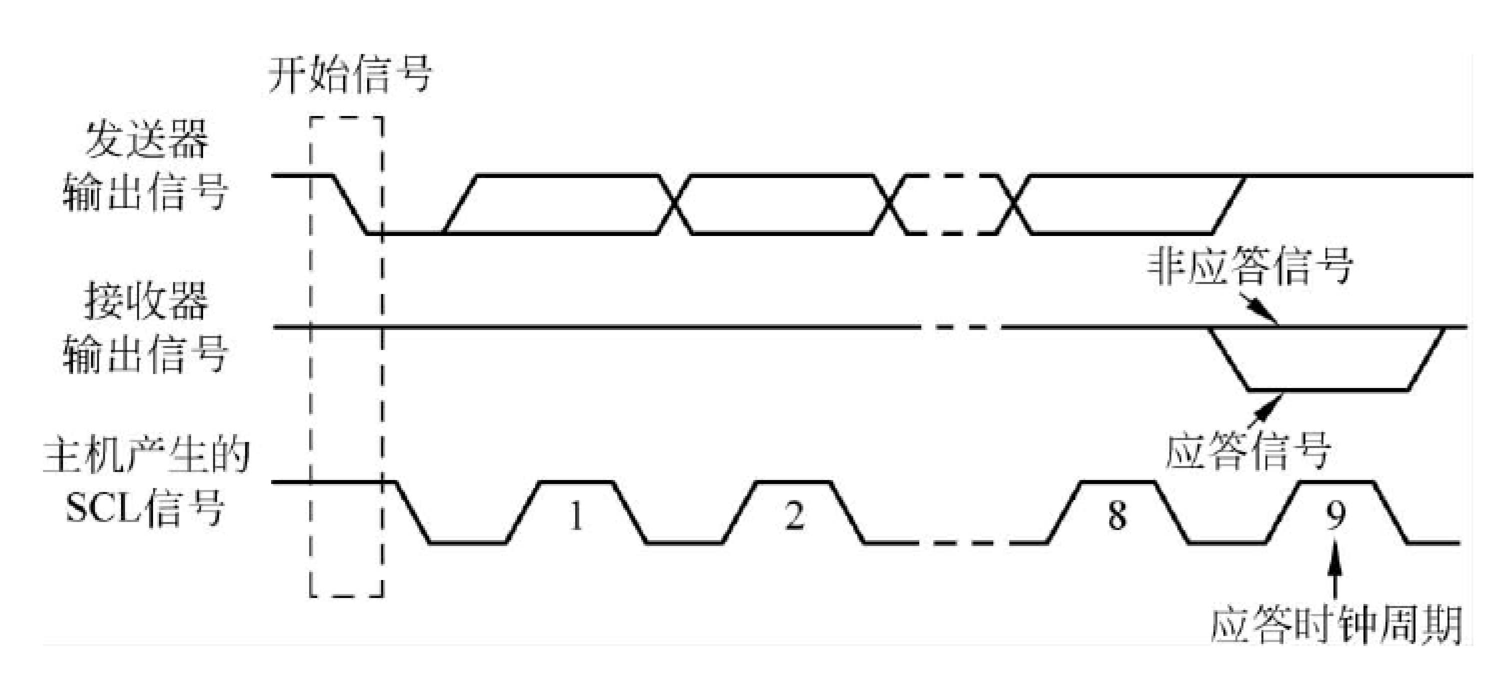


图 11-10　I2C 总线的应答信号

开始、重新开始和停止信号都是由主控制器产生，应答信号由接收器产生，总线上带有 I2C 总线接口的器件很容易检测到这些信号。但是对于不具备这些硬件接口的 MCU 来说，为了能准确地检测到这些信号，必须保证在 I2C 总线的一个时钟周期内对数据线至少进行两次采样。

3）I2C 总线上数据传输格式

一般情况下，一个标准的 I2C 通信由 4 部分组成：开始信号、从机地址传输、数据传输和结束信号。由主机发送一个开始信号，启动一次 I2C 通信，主机对从机寻址，然后在总线上传输数据。I2C 总线上传送的每一个字节均为 8 位，首先发送的数据位为最高位，每传送一个字节后都必须跟随一个应答位，每次通信的数据字节数是没有限制的；在全部数据传送结束后，由主机发送停止信号，结束通信。

如图 11-11 所示，时钟线为低电平时，数据传送将停止进行。这种情况可以用于当接收器接收到一个字节数据后要进行一些其他工作而无法立即接收下个数据时，迫使总线进入等待状态，直到接收器准备好接收新数据时，接收器再释放时钟线使数据传送得以继续正常进行。例如，当接收器接收完主控制器的一个字节数据后，产生中断信号并进行中断处理，中断处理完毕才能接收下一个字节数据，这时，接收器在中断处理时将钳住 SCL 为低电平，直到中断处理完毕才释放 SCL。

4. I2C 总线寻址约定

I2C 总线上的器件一般有两个地址：受控地址和通用广播地址，每个器件有唯一的受控地址用于定点通信，而相同的通用广播地址则用于主控方向时对所有器件进行访问。为了消除 I2C 总线系统中主控器与被控器的地址选择线，最大限度地简化总线连接线，I2C 总线采用了独特的寻址约定，规定了起始信号后的第一个字节为寻址字节，用来寻址被控器件，并规定数据传送方向。

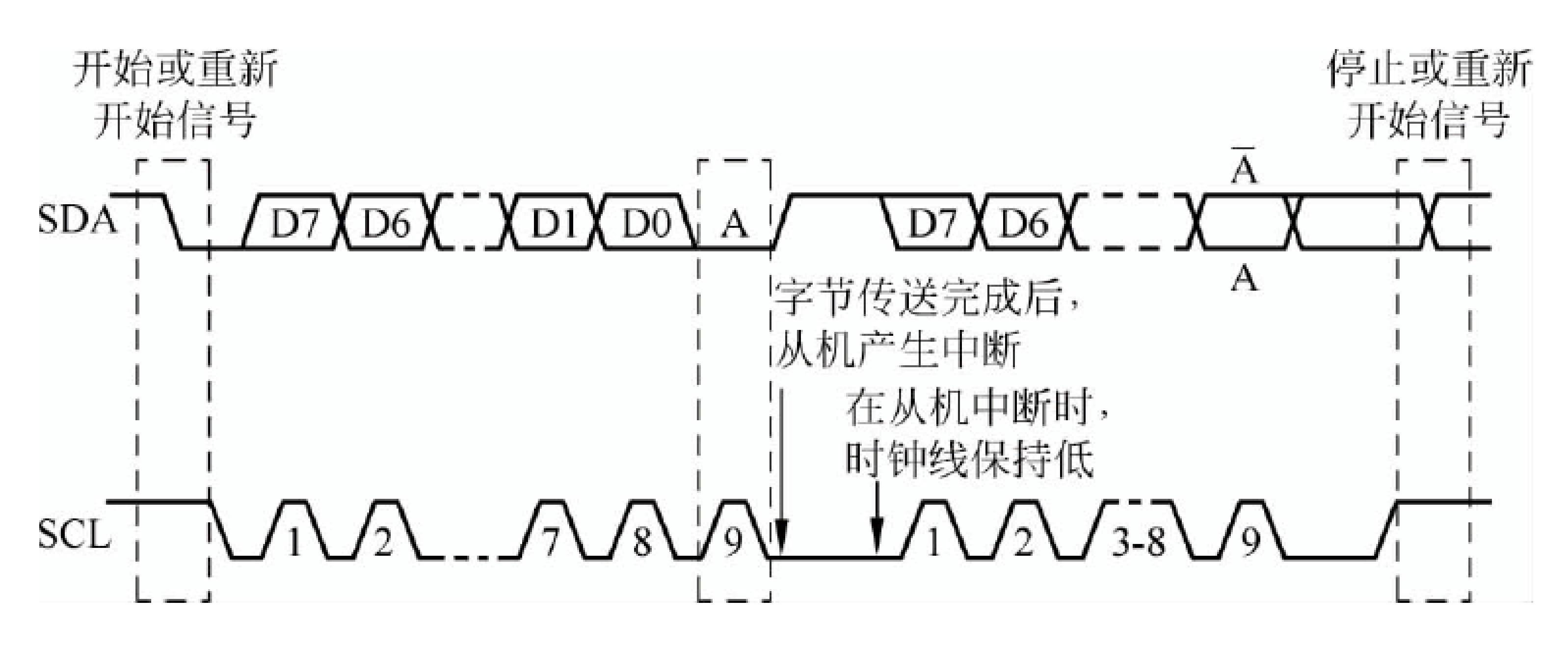


图 11-11　I2C 总线的数据传输格式

在 I2C 总线系统中，寻址字节由被控器的 7 位地址位(D7～D1 位)和一位方向位(D0 位)组成。方向位为 0 时，表示主控器将数据写入被控器，为 1 时表示主控器从被控器读取数据。主控器发送起始信号后，立即发送寻址字节，这时总线上的所有器件都将寻址字节中的 7 位地址与自己的器件地址比较。如果两者相同，则该器件认为被主控器寻址，并发送应答信号，被控器根据数据方向位(R/W)确定自身是作为发送器还是接收器。

MCU 类型的外围器件作为被控器时，其 7 位从机地址在 I2C 总线地址寄存器中设定。而非 MCU 类型的外围器件地址完全由器件类型与引脚电平给定。**I2C 总线系统中，没有两个从机的地址是相同的。**

通用广播地址是用来寻址连接到 I2C 总线上的每个器件，通常在多个 MCU 之间用 I2C 进行通信时使用，可用来同时寻址所有连接到 I2C 总线上的设备。如果一个设备在广播地址时不需要数据，它可以不产生应答来忽略。如果一个设备从通用广播地址请求数据，它可以应答并当作一个从-接收器。当一个或多个设备响应时，主机并不知道有多少个设备应答了。每一个可以处理这个数据的从-接收器可以响应第二个字节。从机不处理这些字节的话，可以响应非应答信号。如果一个或多个从机响应，主机就无法看到非应答信号。通用广播地址的含义一般在第二个字节中指明。

5. 主机向从机读/写一个字节数据的过程

1) 主机向从机写一个字节数据的过程

主机要向从机写一个字节数据时，主机首先产生 START 信号，然后紧跟着发送一个从机地址(7 位)，查询相应的从机，紧接着的第 8 位是数据方向位(R/W)，0 表示主机发送数据(写)，这时候主机等待从机的应答信号(ACK)，当主机收到应答信号时，发送给从机一个位置参数，告诉从机主机的数据在从机接收数组中存放的位置，然后继续等待从机的响应信号，当主机收到响应信号时，发送一个字节的数据，继续等待从机的响应信号，当主机收到响应信号时，产生停止信号，结束传送过程，如图 11-12 所示。

2) 主机从从机读一个字节数据的过程

当主机要从从机读一个字节数据时，主机首先产生 START 信号，然后紧跟着发送一个从机地址，查询相应的从机，注意此时该地址的第 8 位为 0，表明是向从机写命令，这时候主机等待从机的应答信号(ACK)，当主机收到应答信号时，发送给从机一个位置参数，告诉从机主机的数据在从机接收数组中存放的位置，继续等待从机的应答信号，当主机收到应答信号后，主机要改变通信模式(主机将由发送变为接收，从机将由接收变为发送)，所以主机发送重新开始信号，然后紧跟着发送一个从机地址，注意此时该地址的第 8 位为 1，表明将主

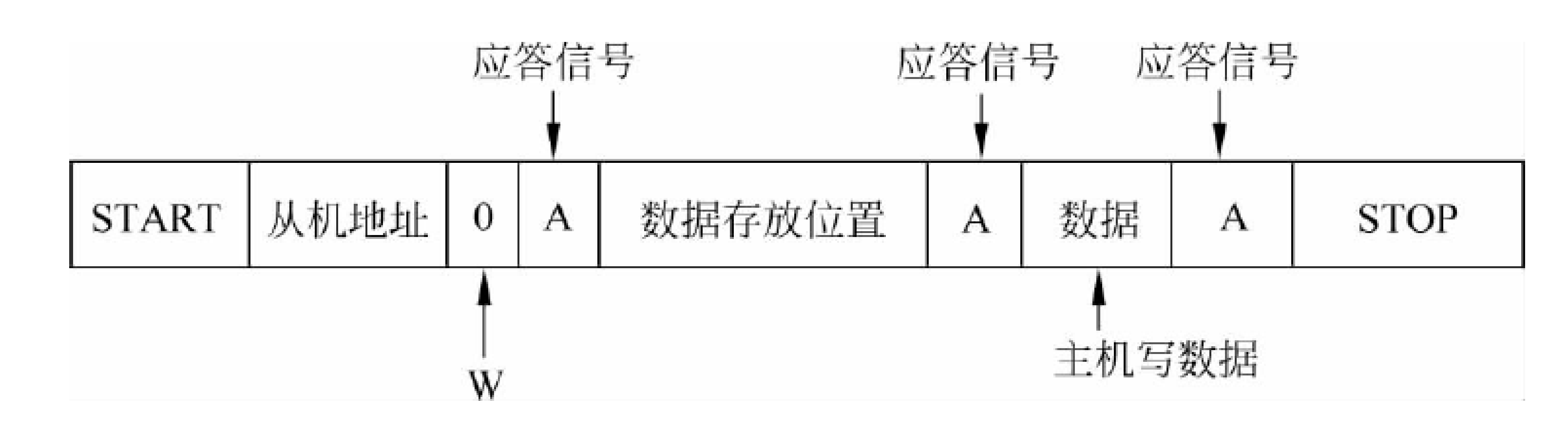


图 11-12　主机向从机写数据

机设置成接收模式开始读取数据，这时主机等待从机的应答信号，当主机收到应答信号时，就可以接收一个字节的数据，当接收完成后，主机发送非应答信号，表示不再接收数据，主机进而产生停止信号，结束传送过程，如图 11-13 所示。

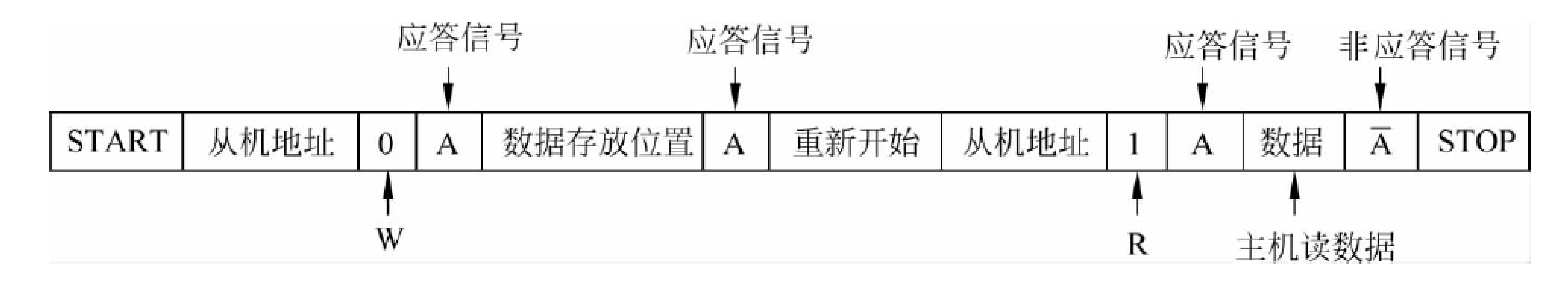


图 11-13　主机从从机读数据

11.2.2　I2C 驱动构件头文件及使用方法

KL25 的 I2C 与通用 I2C 总线规格兼容；多主控(Multimaster)操作；软件可编程 64 种不同的串行时钟频率；软件可选择应答位；中断驱动按位数据传输；带有自动主从模式转换的仲裁丢失中断；呼叫地址判断中断；开始和结束信号产生和检测；重新开始信号产生和检测；应答位的产生和检测；总线被占用检测；一般呼叫识别；10 位扩展地址；支持 System Management Bus (SMBus)；可编程电子脉冲滤波器；与从机地址匹配时从低功率模式被唤醒；可支持扩展从机地址；可支持 DMA。

1. I2C 引脚

80 引脚 KL25 芯片共有 8 组 16 个引脚可配置为 I2C 引脚，其中，PTE24、PTE25、PTA3、PTA4、PTB0、PTB1、PTB2、PTB3 和 PTC8、PTC9 可复用为 I2C0 的相应的 I2C0_SCL，I2C0_SDA 引脚，而 PTE0、PTE1 和 PTA3、PTA4、PTC1、PTC2、PTC10、PTC11 可复用为 I2C1 的相应的 I2C1_SDA，I2C1_SCL 引脚，见表 11-7。

表 11-7　I2C 模块实际使用的引脚

引脚号	引脚名	ALT2	ALT3	ALT4	ALT5	ALT6
24	PTE24		TPM0_CH0		I2C0_SCL	
25	PTE25		TPM0_CH1		2C0_SDA	
43	PTB0	I2C0_SCL	TPM1_CH0			
44	PTB1	I2C0_SDA	TPM1_CH1			
45	PTB2	I2C0_SCL	TPM2_CH0			
46	PTB3	I2C0_SDA	TPM2_CH1			
65	PTC8	I2C0_SCL	TPM0_CH4			

续表

引脚号	引脚名	ALT2	ALT3	ALT4	ALT5	ALT6
66	PTC9	I2C0_SDA	TPM0_CH5			
75	PTE0		UART1_TX	RTC_CLKOUT	CMP0_OUT	I2C1_SDA
76	PTE1	SPI1_MOSI	UART1_RX		SPI1_MISO	I2C1_SCL
29	PTA3	2C1_SCL	TPM0_CH0			
30	PTA4	I2C1_SDA	TPM0_CH1			
56	PTC1	I2C1_SCL		TPM0_CH0		
57	PTC2	I2C1_SDA		TPM0_CH1		

在本例程中将PTC8～9复用为I2C0模块作为主机端，将PTC1～2复用为I2C1模块作为从机端。连接两端对应引脚连线，实现本机两个I2C模块间的通信。

2. I2C驱动构件封装要点分析

I2C模块具有初始化、从从机读取一个字节数据、向从机写一个字节数据、从从机读取N个字节数据、向从机写N个字节数据、开I2C中断、关I2C中断等操作。按照构件化的思想，可将它们封装成独立的功能函数。I2C构件包括头文件I2C.c和I2C.h。I2C构件头文件中主要包括相关宏定义、I2C的功能函数原型说明等内容。I2C构件程序文件的内容是给出I2C各功能函数的实现过程。

在I2C.h中，给出了用于定义所用I2C号的宏定义、I2C所用的引脚组的宏定义。

在I2C.c中，I2c初始化主要用于I2C模块工作的参数设置，如工作时钟、引脚复用配置、模块使能。i2c_read1：从从机读取一个字节数据；i2c_write1：向从机写一个字节数据；i2c_readn：从从机读取N个字节数据；i2c_writen：向从机写N个字节数据以及开I2C中断和关I2C中断函数。

通过以上分析，可以设计I2C构件的几个基本功能函数。

(1) 初始化函数：void i2c_init(uint_8 I2C_No,uint_8 Mode,uint_8 address,uint_8 BaudRate)；

(2) 从从机读取一个字节数据：uint_8 i2c_read1(uint_8 I2C_No,uint_8 DeviceAddr, uint_8 DataLocation, uint_8 *Data)；

(3) 向从机写一个字节数据：uint_8 i2c_write1(uint_8 I2C_No,uint_8 DeviceAddr, uint_8 DataLocation, uint_8 Data)；

(4) 从从机读取N个字节数据：uint_8 i2c_readn(uint_8 I2C_No,uint_8 DeviceAddr, uint_8 DataLocation, uint_8 Data[], uint_8 N)；

(5) 向从机写N个字节数据：uint_8 i2c_writen(uint_8 I2C_No,uint_8 DeviceAddr, uint_8 DataLocation,uint_8 Data[], uint_8 N)；

(6) 使能I2C中断：void i2c_enable_re_int(uint_8 I2C_No)；

(7) 关闭I2C中断：void i2c_disable_re_int(uint_8 I2C_No)；

在I2C构件中，含义相同的参数，它们的命名必须是相同的，这样可增加程序的可读性与易维护性。以上7个基本功能函数的参数说明如表11-8所示。

表 11-8　I2C 基本功能函数参数说明

参　　数	含　　义	备　　注
I2C_No	模块号	No＝0，表示 I2C0；No＝1，表示 I2C1
Mode	I2C 主从机选择	Mode＝0，设为从机；Mode＝1，设为主机
address	本模块初始化地址	寻址字节由 7 位地址位和一位方向位组成，地址范围为 1～255
BaudRate	波特率	其单位为 kb/s，其取值为 50，75，100，150，300
DeviceAddr	设备地址	对应从机的地址，范围为 1～255
DataLocation	数据在从机接收数组中的位置	从机定义一个 8 位的全局数组，用于接收主机发来的数据，范围为 0～255
* Data	一个字节数据	带回收到的一个字节数据
Data	一个字节数据	要发给从机的一个字节数据
Data[]	数组的首地址	读出数据的缓冲区或要写入的数据首地址
N	从从机读的字节个数	范围为 1～255

为增加 I2C 构件的可移植性，需要将 I2C 使用到的并与芯片相关的变量进行宏定义。这使得 I2C 构件移植到其他型号芯片上时，只需修改对应的宏定义即可，无须修改 i2c.c 文件中的程序。i2c.h 文件宏定义及其含义如表 11-8 所示。

3. I2C 驱动构件头文件

```
//===========================================
//文件名称：i2c.h
//功能概要：i2c 底层驱动构件头文件
//版权所有：苏州大学恩智浦嵌入式中心(sumcu.suda.edu.cn)
//更新记录：2013-03-12    V1.2
//            2016-03-17    V3.0
//===========================================
#ifndef _I2C_H                              //防止重复定义(开头)
#define _I2C_H
#include "common.h"                         //包含公共要素头文件
//模块宏定义

//IIC 号的宏定义
#define IIC_0         0                     //I2C0
#define IIC_1         1                     //I2C1

//根据串口实际硬件引脚，确定以下宏常量值
//在此工程中，只使用 IIC0 组中的第 4 个，IIC1 组中的第 3 个，
//因此在此只需要将 IIC_0_GROUP 宏定义为 4，IIC_1_GROUP 宏定义为 3

//IIC_0：1=PTE24～25 脚，2=PTB0～1 脚，3=PTB2～3 脚，4=PTC8～9 脚
#define IIC_0_GROUP 4                       //SD-FSL-KL25-EVB 板上使用 PTC8～9 脚

//IIC_1：1=PTE1～0 脚，2=PTA3～4 脚，3=PTC1～2 脚，4=PTC10～11 脚
#define IIC_1_GROUP 3                       //SD-FSL-KL25-EVB 板上使用 PTC1～2 脚
```

```
// 功能接口(i2c 通信函数声明)
//===========================================================
//函数名称：i2c_init
//功能概要：初始化 IICX 模块
//参数说明：I2C_No：模块号，其取值为 0，1
//          Mode 模式   1：主机     0：从机
//          address 本模块初始化地址范围 1~255
//          BaudRate 为波特率，其单位为 kb/s，其取值为 50，75，100，150，300
//函数返回：无
//===========================================================
void i2c_init(uint_8 I2C_No,uint_8 Mode,uint_8 address,uint_8 BaudRate);

//===========================================================
//函数名称：i2c_read1
//功能概要：从从机读 1 个字节数据
//参数说明：I2C_No：模块号，其取值为 0，1
//          DeviceAddr：设备地址范围 1~255
//          DataLocation：数据在从机接收数组中的存放位置范围 0~255
//          Data：带回收到的一个字节数据
//函数返回：为 0，成功读一个字节；为 1，读一个字节失败
//函数说明：内部调用 send_signal，wait
//===========================================================
uint_8 i2c_read1(uint_8 I2C_No,uint_8 DeviceAddr, uint_8 DataLocation,uint_8 * Data);

//===========================================================
//函数名称：i2c_write1
//功能概要：向从机写一个字节数据
//参数说明：I2C_No：模块号，其取值为 0，1
//          DeviceAddr：设备地址范围 1~255
//          DataLocation：数据在从机接收数组中的存放位置范围 0~255
//          Data：要发给从机的一个字节数据
//函数返回：为 0，成功写一个字节；为 1，写一个字节失败
//函数说明：内部调用 send_signal，wait
//===========================================================
uint_8 i2c_write1(uint_8 I2C_No,uint_8 DeviceAddr, uint_8 DataLocation,uint_8 Data);

//===========================================================
//函数名称：i2c_readn
//功能概要：从从机读 N 个字节数据
//参数说明：I2C_No：模块号，其取值为 0，1
//          DeviceAddr：设备地址范围 1~255
//          DataLocation：数据在从机接收数组中的存放位置范围 0~255
//          Data：读出数据的缓冲区
//          N：从从机读的字节个数
//函数返回：为 0，成功读 N 个字节；为 1，读 N 个字节失败
//函数说明：内部调用 i2c_read1
//===========================================================
uint_8 i2c_readn(uint_8 I2C_No,uint_8 DeviceAddr, uint_8 DataLocation, \
                                           uint_8 Data[], uint_8 N);
```

```
//=======================================================================
//函数名称: i2c_writen
//功能概要: 向从机写 N 个字节数据
//参数说明: I2C_No: 模块号,其取值为 0,1
//          DeviceAddr: 设备地址范围 1～255
//          DataLocation: 数据在从机接收数组中的存放位置范围 0～255
//          Data: 要写入的数据的首地址
//          N: 从从机读的字节个数
//函数返回: 为 0,成功写 N 个字节;为 1,写 N 个字节失败
//函数说明: 内部调用 i2c_write1
//=======================================================================
uint_8 i2c_writen(uint_8 I2C_No,uint_8 DeviceAddr, uint_8 DataLocation,\
                                                  uint_8 Data[], uint_8 N);
//=======================================================================
//函数名称: i2c_re_enable_int.
//功能说明: 打开 i2c 的 IRQ 中断
//函数参数: i2cNO:i2c 模块号,其取值为 0,1
//函数返回: 无
//=======================================================================
void i2c_enable_re_int(uint_8 I2C_No);

//=======================================================================
//函数名称: i2c_re_disable_int.
//功能说明: 关闭 i2c 的 IRQ 中断
//函数参数: i2cNO:i2c 模块号,其取值为 0,1
//函数返回: 无
//=======================================================================
void i2c_disable_re_int(uint_8 I2C_No);
#endif  //防止重复定义(结尾)
```

4. I2C 驱动构件使用方法

在 I2C 驱动构件的头文件(i2c.h)中包含的内容有初始化 I2C 模块(i2c_init)、从从机读取一个字节数据(i2c_read1)、向从机写一个字节数据(i2c_write1)、从从机读取 *N* 个字节数据(i2c_readn)、向从机写 *N* 个字节数据(i2c_writen)、使能 I2C 中断(i2c_enable_re_int)、关闭 I2C 中断(i2c_disable_re_int)。

下面介绍构件的使用方法,举例如下。

1) 主机

(1) 在主函数 main 中,初始化 I2C 模块。第一个参数为 I2C 的模块号,第二个参数为主机还是从机,第三个参数为模块初始化地址,第四个为波特率。

```
i2c_init(IIC_0,1,MasterAddress,100);        //第四个参数为波特率,其单位为 kb/s
```

(2) 声明一个数组用于储存向从机发送的数据,并赋值。

```
uint_8 data[12];                          //发向从机的数据;
strcpy(data,"Version3.4\n");              //为 data 数组赋值
```

(3) 在主循环中,小灯每闪烁一次,向从机发送一个字节数据。

```
//依次向从机写 data 中数据,0x73 为从机地址,0x02 为数据在从机接收数组中的位置
i2c_write1(IIC_0, 0x73, 0x02, data[Num_flag]);
```

2) 从机

(1) 在主函数 main 中,初始化 I2C 模块。第一个参数为 I2C 的模块号,第二个参数为主机还是从机,第三个参数为模块初始化地址,第四个参数为波特率。

```
i2c_init(IIC_1,0,0x73,75);    //i2c1 模块初始化
```

(2) 因为 I2C_1 初始化为从机,需要接收从主机发来的数据,所以需要使能模块中断。

```
i2c_enable_re_int(IIC_1);    //使能模块中断
```

(3) 在 I2C1 中断服务程序 I2C1_IRQHandler 中,用于接收从主机发来的数据。当主机发送的地址与从机的默认地址匹配时,从机用一个全局数组 buf[]来接收主机发送来的数据。

接收的第一个数为主机的数据在数组中的位置,接收之后赋值给变量 visitaddr:

```
visitaddr=I2C0_D;
```

接收的第二个数为主机发送过来的数据,把它赋值给中间变量 data:

```
data=I2C0_D;
```

当地址匹配,位置和数据都接收成功之后,把数据放在 buf[]数组中:

```
buf[visitaddr]=data;
```

(4) 然后可以通过串口把数据发送到 PC:

```
uart_send1(UART_TEST,buf[visitaddr]);    //发送主机传送过来的数据
```

5. I2C 驱动构件测试实例

为使读者直观地了解 I2C 模块之间传输数据的过程,I2C 驱动构件测试实例使用串口将 I2C0 和 I2C1 模块之间传输的数据显示在 PC 上。测试工程位于网上教学资源中的"..\KL25-program\ch11-KL25-SPI-I2C-TSI"文件夹中,硬件连接见工程文档。测试工程功能如下。

(1) 使用串口 1 与外界通信,波特率为 9600,1 位停止位,无校验。

(2) 初始化 I2C0 和 I2C1,I2C0 模块作为主机,I2C1 模块作为从机。

(3) 启动 I2C1 接收中断,将 I2C1 接收到的数据通过串口发送到 PC。

(4) 在 main.c 的主循环中 I2C0 向 I2C1 发送字符串"Version3.4",I2C1 通过中断接收到这些字符并判断,并通过串口发送给 PC。

(5) 复位之后通过串口 1 输出"This is iic Test!"。

11.2.3　I2C 模块的编程结构

I2C 的两个模块，所有的终端用户可以访问的 I2C 寄存器共有 22 个，每个模块各有 11 个 8 位寄存器，但每个模块常用的只有 5 个，这些寄存器复位为 0。表 11-9 给出了 I2C 的模块 0 和模块 1 常用的 10 个寄存器，每个寄存器均可"读/写"。只要理解和掌握这 10 个寄存器的用法，了解 I2C 总线协议，就可以进行 I2C 模块的底层驱动构件设计了。这里只讲述 I2C0，对 I2C1 的应用可参照 I2C0。

表 11-9　I2C 的模块 0 和模块 1 常用的 10 个寄存器

模块号	寄存器名称	缩写	地址	基本功能
0	地址寄存器	I2C0_A1	0x4006_6000	设置从机地址
	分频寄存器	I2C0_F	0x4006_6001	设置 I2C 模块的工作频率
	控制寄存器 1	I2C0_C1	0x4006_6002	设置传输格式、中断使能
	状态寄存器	I2C0_S	0x4006_6003	表明 I2C 模块的工作状态
	数据寄存器	I2C0_D	0x4006_6004	收发数据
1	地址寄存器	I2C1_A1	0x4006_7000	设置从机地址
	分频寄存器	I2C1_F	0x4006_7001	设置 I2C 模块的工作频率等
	控制寄存器 1	I2C1_C1	0x4006_7002	设置传输格式、中断使能等
	状态寄存器	I2C1_S	0x4006_7003	表明 I2C 模块的工作状态
	数据寄存器	I2C1_D	0x4006_7004	收发数据

1. I2C 地址寄存器

该寄存器包含被 I2C 模块调用的从机的地址，结构如表 11-10 所示。复位之后为 0。

表 11-10　I2C0_A1 结构

数据位	D7～D1	D0
读	AD[7:1]	0
写		—

D7～D1(AD[7:1])——当作为从机被寻址时，此位段包含 I2C 模块所使用的初始从机地址。此位段用于 7 位地址方案和 10 位地址方案中的低 7 位。

D0(0)——保留位。这个只读位段是保留的，读取值为 0。

2. I2C 分频寄存器

该寄存器主要用于倍频因子和时钟频率的设置，结构如表 11-11 所示。复位之后为 0。

表 11-11　I2C0_F 结构

数据位	D7、D6	D5～D0
读/写	MULT	ICR

D7、D6(MULT)——定义倍频因子 MUL。此因子与 SCL 分频一起使用，以产生 I2C 波特率。当 MUL 分别为 00、01、02 时，倍频因子的值分别为 1、2 和 4。

D5～D0(ICR)——时钟频率。此位段和 MULT 位段确定 I2C 波特率、SDA 保持时间、

SCL 开始保持时间和 SCL 停止保持时间。SCL 分频因子乘以倍频因子(MUL)决定了 I2C 波特率。I2C 波特率=总线频率(Hz)/(MUL×SCL 分频因子)。SDA 保持时间是从 I2C 时钟(SCL)的下降沿到 I2C 数据(SDA)变化的延迟时间。SDA 保持时间=总线周期(s)×MUL×SDA 保持值。SCL 开始保持时间是当 SCL 为高电平(启动条件)时,从 SDA 的下降沿(I2C 数据)到 I2C 时钟(SCL)的下降沿的延迟。SCL 开始保持时间=总线周期(s)×MUL×SCL 开始保持值。SCL 停止保持时间是当 SCL 为高电平(停止条件)时,从 I2C 时钟(SCL)的上升沿到 SDA 的上升沿(I2C 数据)的延迟。SCL 停止保持时间=总线周期(s)×MUL×SCL 停止保持值。例如,如果总线频率为 8MHz,表 11-12 给出了选取不同的 ICR 和 MULT,I2C 波特率为 100kb/s 的对应保持时间值。

表 11-12 MULT、ICR 和保持时间值

MULT	ICR	保持时间/μs		
		SDA	SCL Start	SCL Stop
2h	00h	3.500	3.000	5.500
1h	07h	2.500	4.000	5.250
1h	0Bh	2.250	4.000	5.250
0h	14h	2.125	4.250	5.125
0h	18h	1.125	4.750	5.125

3. I2C 控制寄存器 1

该寄存器主要用于 I2C 的使能、I2C 中断使能、传送模式选择、DMA 使能等的设置,复位之后为 0,如表 11-13 所示。

表 11-13 I2C0_C1 结构

数据位	D7	D6	D5	D4	D3	D2	D1	D0
读/写	IICEN	IICIE	MST	TX	TXAK	RSTA	WUEN	DMAEN
复位	0							

D7(ICEN)——I2C 使能。使能 I2C 模块的操作。0 禁用,1 使能。

D6(IICEN)——I2C 中断使能。使能 I2C 中断请求。0 禁用,1 使能。

D5(MST)——主机模式选择。当 MST 位从 0 改变为 1 时,在总线上产生一个开始信号并且主模式被选中。当该位从 1 变到 0 时,产生一个停止信号并且操作模式由主机切换到从机。0 从机模式,1 主机模式。

D4(TX)——传送模式选择。选择主机和从机的传输方向。在主机模式下,该位根据需要传输的类型来设置。因此,处于地址周期,该位总是置位。当作为从机被寻址时,该位必须根据状态寄存器中的 SRW 位通过软件设置。0 接收,1 发送。

D3(TXAK)——传送应答使能。在从机和主机接收器的应答周期中,指定的数据被驱动到 SDA 上。FACK 位的值影响 NACK/ACK 的产生。注意:SCL 保持低电平直到 TXAK 被写入。当 TXAK 为 0 时,应答信号在接收完一个字节后(如果 FACK 被清零)或当前正在接收字节时(如果 FACK 置位)被发送到总线上。当 TXAK 为 1 时,没有应答信号在接收完一个字节后(如果 FACK 被清零)或当前正在接收字节时(如果 FACK 置位)被

发送到总线上。

D2(RSTA)——重复开始。写 1 到该位为当前的主机提供了一个产生重启的条件。该位读取值总是 0。在错误的时间尝试重启会导致仲裁丢失。

D1(WUEN)——唤醒使能。当从机地址匹配发生时,I2C 模块能够在没有外围总线运行的情况下从低功耗模式中唤醒 CPU。当 WUEN 为 0 时,正常运行。在低功耗模式下发生地址匹配时无中断产生。当 WUEN 为 1 时,在低功耗模式下使能唤醒功能。

D0(DMAEN)——DMA 使能。DMAEN 位使能或禁用 DMA 功能。当 DMAEN 为 0 时禁用所有的 DMA 信令。当 DMAEN 为 1 时,DMA 传送被使能,并且满足下列条件时将触发 DMA 请求:①当 FACK=0 时,一个数据字节被接收,其要么是地址要么是数据被传送(ACK / NACK 自动)。②当 FACK=0 时,接收到的第一个字节匹配 A1 寄存器的内容或者其是广播地址。如果有地址匹配发生,那么 IAAS 和 TCF 被置位。如果传输方向是已知的由主机到从机,那么它不需要检查 SRW。基于这个假设,DMA 也可以在这种情况下使用。在其他情况下,如果主机读取从机的数据,那么就需要重写 C1 寄存器操作。基于这个假设,DMA 不能被用。③当 FACK=1 时,一个地址或数据字节被传送。

4. I2C 状态寄存器

I2C0_S 寄存器主要用于 I2C 模块的传输完成的判断和设置,以及中断挂起的设置,如表 11-14 所示。D0～D6 复位之后为 0,D7 复位后为 1。

表 11-14　I2C0_S 结构

数据	D7	D6	D5	D4	D3	D2	D1	D0
读	TCF	IAAS	BUSY	ARBL	RAM	SRW	IICIF	RXAK
写	—		—	W1C		—	W1C	—
复位	1	0						

D7(TCF)——传输完成标志。该位在一个字节和应答信号传输完成时被置位。该位仅在传送到 I2C 模块或者从 I2C 模块传送出去期间或之后有效。在接收模式下读取 I2C 数据寄存器清零 TCF 位,或者在传送模式下写 I2C 数据寄存器清零它。0 表示正在传送,1 表示传送完毕。

D6(IAAS)——作为从机被寻址。满足下列条件之一时,此位被置位:①呼叫地址要与 A1 的寄存器中的可编程从机初始地址或 RA 扩展寄存器中的地址(其必须设置为非零值)相匹配。②GCAEN 被置位并且广播被接收。③SIICAEN 被置位并且广播地址匹配第二次可编程的从机地址。④ALERTEN 被置位并且 SMBus 警告响应地址被接收。⑤RMEN 被置位并且地址被接收(其值的范围在 A1 和 RA 寄存器之间)。

该位在 ACK 位之前被设置。CPU 必须检查 SRW 位并设置相应的 TX/RX。写 C1 寄存器清零该位。当 IAAS 为 0 时,没有被寻址。当 IAAS 为 1 时,作为从机被寻址。

D5(BUSY)——总线忙。无论主机从机模式,该位总表示总线的状态。检测到 START 号时该位被置位,检测到 STOP 信号时,该位被清 0。0 表示总线空闲,1 表示总线忙。

D4(ARBL)——仲裁丢失。当仲裁程序丢失时,此位由硬件置位。ARBL 位必须由软

件写1来清零。当ARBL为0时,为标准总线操作;当ARBL为1时为仲裁丢失。

D3(RAM)——扩展地址匹配。满足下列的任何条件时此位被设置为1:①接收任何与RA寄存器中地址相匹配的非零的广播地址;②RMEN位被置位并且广播地址的值在A1和RA寄存器的值的范围内。**注意**:为了RAM位能被正确地设置为1,C1[IICIE]必须设置为1。写C1寄存器清除该位为0。当RAM为0时没有被寻址,当RAM为1时作为从机被寻址。

D2(SRW)——从机读/写。当作为一个从机时,SRW显示发送到主机的广播地址读/写命令位的值。当SRW为0时从机接收,主机写入到从机。当SRW为1时从机发送,主机读取从机。

D1(IICIF)——中断标志。当中断被挂起时该位置位。比如在中断程序中,该位必须通过软件写1来进行清零。以下事件可以置位该位:①在接收模式下通过写0或者写1设置该位后,如果FACK是0,那么一个字节(包括ACK/NACK位)传输完成后,一个ACK或NACK被发送到总线上。②如果FACK是1,一个字节传输完成(不包括ACK/NACK位)。③从机地址匹配广播地址(包括初始从机地址、扩展从机地址、警告响应地址、第二个从机地址或广播)。④仲裁丢失。⑤在SMBus模式下,任何超时(除了SCL和SDA高超时外)。⑥如果STOPIE位在输入干扰滤波器寄存器中为1,I2C总线停止检测。注意:为了清除I2C总线停止检测中断:在中断服务例程中,首先通过写1清零输入电子脉冲滤波寄存器中的STOPF位,然后清除IICIF位。如果此序列反转,IICIF位再次被声明。当IICIF为0时没有中断挂起,当IICIF为1时有中断挂起。

D0(RXAK)——接收应答。当RXAK为0时在总线上的一个字节的数据传输完成后,应答信号被接收;当RXAK为1时没有应答信号被检测到。

5. I2C数据I/O寄存器

在主机传送模式,当数据被写入到这个寄存器后,数据传输开始。首先发送最高位。在主机接收模式下,读取该寄存器开始接收下一个字节的数据。注意:要改变主机接收模式的话,那么在读数据寄存器之前必须改变I2C模式以防止不慎启动主机,把接收的数据传输出去。在从机模式下,同样的功能都可以在地址匹配发生后获得。在主模式和从模式下的传输开始时,C1[TX]位必须正确地反映所需的传输方向。例如,如果I2C模块配置为主机传送,但是却需要主机接收,那么读取数据寄存器将不会启动接收。当I2C模块在主机接收或从机接收模式被配置,那么读数据寄存器返回最后一个接收到的字节。数据寄存器不反映I2C总线上传输的每个字节,也不可以通过读回它来验证一个字节是否被正确写入到数据寄存器。在主机传送模式下,写入到数据寄存器中的数据的第一个字节跟随MST(起始位)或RSTA的声明(重复开始位)被用于地址传输并且必须由广播地址(7~1位)和所需的读/写位(0位)连接组成。

11.2.4 I2C驱动构件的设计

1. I2C基本编程步骤

实现I2C间的数据传输主要涉及以下几个寄存器:地址寄存器(I2C0_A1)、分频寄存器(I2C0_F)、控制寄存器1(I2C0_C1)、状态寄存器(I2C0_S)、数据I/O寄存器(I2Cx_D)。其中,地址寄存器I2C0_A1用于设置I2C模块调用的从机的地址;分频寄存器I2C0_F用

于倍频因子和时钟频率的设置；控制寄存器 1I2C0_C1 用于 I2C 的使能、I2C 中断使能、传送模式选择、DMA 使能等的设置；状态寄存器 I2C0_S 用于 I2C 模块的传输完成的判断和设置以及中断挂起的设置。基本编程步骤(以主机为例)如下。

(1) 打开 I2C 模块时钟源，SIM_SCG4_I2C0。

(2) 引脚复用为 I2C0 功能。SCL 使用 PTC8，SDA 使用 PTC9。

(3) 配置控制寄存器 1(I2C0_C1)，使能 I2C 模块(C1 [IICEN]＝1)，传送应答使能(C1 [TXAK]＝1)。

(4) 配置分频寄存器(I2C0_F)，配置倍频因子和时钟频率。

(5) 配置地址寄存器(I2C0_A1)，设定本机作为从机时的默认地址。

2. I2C 驱动构件源码

```
//========================================================================
//文件名称：i2c.c
//功能概要：i2c 底层驱动构件源文件
//版权所有：苏州大学恩智浦嵌入式中心(sumcu.suda.edu.cn)
//更新记录：2013-05-4    V2.1
//            2016-03-18    V3.0
//========================================================================
#include  "i2c.h"

static const uint_8 data[]={300,0x14,150,0x54,100,0x1f,75,0x25,50,0x5f};
static void send_signal(uint_8 Signal,uint_8 I2C_No);
static uint_8 wait(uint_8 x,uint_8 I2C_No);
static I2C_MemMapPtr i2c_get_base_address(uint_8 I2C_No);

//========================================================================
//函数名称：send_signal
//功能概要：根据需要产生开始或停止信号
//参数说明：I2C_No：模块号，其取值为 0,1
//           Signal：'S'(Start)，产生开始信号'O'(Over)， 产生停止信号
//函数返回：无
//========================================================================
void send_signal(uint_8 Signal,uint_8 I2C_No)
{
    //获取 i2c 模块基址
    I2C_MemMapPtr num = i2c_get_base_address(I2C_No);
    if(num==I2C0)
    {
        if (Signal == 'S')
        {
            //i2c0_c 主机模式选择位 MST 由 0 变为 1，可以产生开始信号
            BSET(I2C_C1_MST_SHIFT,I2C0_C1);
        }
        else if (Signal == 'O')
        {
            //主机模式选择位 MST 由 1 变为 0，可以产生停止信号
            BCLR(I2C_C1_MST_SHIFT,I2C0_C1);
```

```
            }
        }
        else if(num==I2C1)
        {
            if (Signal == 'S')
            {
                //i2c0_c 主机模式选择位 MST 由 0 变为 1,可以产生开始信号
               BSET(I2C_C1_MST_SHIFT,I2C1_C1);
             }
             else if (Signal == 'O')
            {
                //主机模式选择位 MST 由 1 变为 0,可以产生停止信号
                BCLR(I2C_C1_MST_SHIFT,I2C1_C1);
             }
        }
}
//==========================================================
//函数名称: wait
//功能概要: 在时限内,循环检测接收应答标志位,或传送完成标志位,判断 MCU
//          是否接收到应答信号或一个字节是否已在总线上传送完毕
//参数说明: I2C_No: 模块号,其取值为 0,1
//   x:x = 'A'(Ack),等待应答;x = 'T'(Transmission),等待一个字节数据传输完成
//函数返回: 0: 收到应答信号或一个字节传送完毕;
//          1: 未收到应答信号或一个字节没传送完
//==========================================================
uint_8 wait(uint_8 x,uint_8 I2C_No)
{
    uint_16 ErrTime, i;
    //获取 i2c 模块基地址
    I2C_MemMapPtr num = i2c_get_base_address(I2C_No);
    ErrTime = 255 * 10;                 //定义查询超时时限
    for (i = 0;i < ErrTime;i++)
    {
        if (x == 'A')                   //等待应答信号
        {
            if((I2C_S_REG(num) & I2C_S_RXAK_MASK)==0)
                return 0;               //传送完一个字节后,收到了从机的应答信号
         }
        else if (x == 'T')              //等待传送完成一个字节信号
        {
           if ((I2C_S_REG(num) & I2C_S_IICIF_MASK) != 0)
           {
               //清 IICIF 标志位
                (I2C_S_REG(num) |=(0 | I2C_S_IICIF_MASK));
                    return 0;           //成功发送完一个字节
            }
         }
     }
    if (i >= ErrTime)
```

```
        return 1;                //超时,没有收到应答信号或发送完一个字节
}

//==========================================
//函数名称: i2c_init
//功能概要: 初始化 IICX 模块
//参数说明: I2C_No: 模块号,其取值为 0,1
//          Mode 模式  1: 主机    0: 从机
//          address 本模块初始化地址范围 1~255
//          BaudRate 为波特率,其单位为 kb/s,其取值为 50,75,100,150,300
//函数返回: 无
//==========================================
void i2c_init(uint_8 I2C_No,uint_8 Mode,uint_8 address,uint_8 BaudRate)
{
    //获取 I2C 模块的基址
    uint_8 index;
    I2C_MemMapPtr num = i2c_get_base_address(I2C_No);
    if(I2C_No<0||I2C_No>1)                //如果模块号错误则强制其为 0
    {
        I2C_No=0;
    }
    if(num==I2C0)
    {
        //I2C0 Clock Gate Control --enable
        BSET(SIM_SCGC4_I2C0_SHIFT,SIM_SCGC4);
        I2C0_C1=0X00;
        BSET(I2C_S_IICIF_SHIFT,I2C0_S);

        //IIC_0: 1=PTE24~25 脚,2=PTB0~1 脚,3=PTB2~3 脚,4=PTC8~9 脚
        //引脚复用为 IIC0 功能
        #if (IIC_0_GROUP == 1)
        PORTE_PCR24 = PORT_PCR_MUX(0x5);    //使能 IIC0_SCL
        PORTE_PCR25 = PORT_PCR_MUX(0x5);    //使能 IIC0_SDA
        #endif

        #if (IIC_0_GROUP == 2)
        PORTB_PCR0 = PORT_PCR_MUX(0x2);     //使能 IIC0_SCL
        PORTB_PCR1 = PORT_PCR_MUX(0x2);     //使能 IIC0_SDA
        #endif

        #if (IIC_0_GROUP == 3)
        PORTB_PCR2 = PORT_PCR_MUX(0x2);     //使能 IIC0_SCL
        PORTB_PCR3 = PORT_PCR_MUX(0x2);     //使能 IIC0_SDA
        #endif

        #if (IIC_0_GROUP == 4)
        PORTC_PCR8 = PORT_PCR_MUX(0x2);     //使能 IIC0_SCL
        PORTC_PCR9 = PORT_PCR_MUX(0x2);     //使能 IIC0_SDA
        #endif
```

```
        //设置 MULT 和 ICR,kL25 的 MCU 总线频率为 24MHz,在总线上分频得 75kb/s
        I2C0_A1=address;                          //本机作为从机时的默认地址
        for(index=0;index<10;index+=2)
        {
            if(data[index]==BaudRate)
            {
                index++;
                I2C0_F=data[index];               //将定义的参数赋给 I2C1_F
                break;
            }
        }

        if(1 == Mode)                             //主机模式
        {
            BSET(I2C_C1_IICEN_SHIFT,I2C0_C1); //开 i2c0 模块使能
            BSET(I2C_C1_TXAK_SHIFT,I2C0_C1); //置位 i2c0 TXAK
        }
        else                                      //从机模式
        {
            BSET(I2C_C1_IICEN_SHIFT,I2C0_C1); //使能 I2C 模块
            BSET(I2C_C1_IICIE_SHIFT,I2C0_C1);  //来 I2C 中断
            BSET(I2C_C1_MST_SHIFT,I2C0_C1);    //设置成主机模式
            BCLR(I2C_C1_TX_SHIFT,I2C0_C1);     //TX = 0,MCU 设置为接收模式
//              i=I2C0_D;                         //读出 IIC1D,准备接收数据
            BCLR(I2C_C1_MST_SHIFT,I2C0_C1);    //MST 位由 1 变成 0,设置为从机模式
        }

    }
    else
    {
        //I2C1 Clock Gate Control --enable
        BSET(SIM_SCGC4_I2C1_SHIFT,SIM_SCGC4);
        //SIM_SCGC4 |=SIM_SCGC4_I2C1_MASK;
        I2C1_C1=0X00;
        BSET(I2C_S_IICIF_SHIFT,I2C1_S);

        //IIC_1: 1=PTE1~0 脚,2=PTA3~4 脚, 3=PTC1~2 脚, 4=PTC10~11 脚
        //SD-FSL-KL25-EVB 板上使用 PTC1~2 脚
        #if (IIC_1_GROUP == 1)
        PORTE_PCR1 = PORT_PCR_MUX(0x6);     //使能 IIC1_SCL
        PORTE_PCR0 = PORT_PCR_MUX(0x6);     //使能 IIC1_SDA
        #endif

        #if (IIC_1_GROUP == 2)
        PORTA_PCR3 = PORT_PCR_MUX(0x2);     //使能 IIC1_SCL
        PORTA_PCR4 = PORT_PCR_MUX(0x2);     //使能 IIC1_SDA
        #endif

        #if (IIC_1_GROUP == 3)
        PORTC_PCR1 = PORT_PCR_MUX(0x2);     //使能 IIC1_SCL
```

```
        PORTC_PCR2 = PORT_PCR_MUX(0x2);       //使能 IIC1_SDA
        #endif

        #if (IIC_1_GROUP == 4)
        PORTC_PCR10 = PORT_PCR_MUX(0x2);      //使能 IIC1_SCL
        PORTC_PCR11 = PORT_PCR_MUX(0x2);      //使能 IIC1_SDA
        #endif

        I2C1_A1=address;                      //本机作为从机时的默认地址
        for(index=0;index<10;index+=2)
        {
            if(data[index]==BaudRate)
            {
                index++;
                I2C1_F=data[index];           //将定义的参数赋给 I2C1_F
                break;
            }
        }

        if(1 == Mode)                         //主机模式
        {
            BSET(I2C_C1_IICEN_SHIFT,I2C1_C1); //开 i2c0 模块使能
            BSET(I2C_C1_TXAK_SHIFT,I2C1_C1);  //置位 i2c0 TXAK
        }
        else                                  //从机模式
        {
            BSET(I2C_C1_IICEN_SHIFT,I2C1_C1); //使能 I2C 模块
            BSET(I2C_C1_IICIE_SHIFT,I2C1_C1); //来 I2C 中断
            BSET(I2C_C1_MST_SHIFT,I2C1_C1);   //设置成主机模式
            BCLR(I2C_C1_TX_SHIFT,I2C1_C1);    //TX = 0,MCU 设置为接收模式
//              i=I2C0_D;                     //读出 IIC1D,准备接收数据
            BCLR(I2C_C1_MST_SHIFT,I2C1_C1);   //MST 位由 1 变成 0,设置为从机模式
        }
    }
}

//==========================================================================
//函数名称：i2c_read1
//功能概要：从从机读一个字节数据
//参数说明：I2C_No：模块号,其取值为 0,1
//          DeviceAddr：设备地址范围 1～255
//          DataLocation：数据在从机接收数组中的存放位置范围 0～255
//          Data：带回收到的一个字节数据
//函数返回：为 0,成功读一个字节;为 1,读一个字节失败
//函数说明：内部调用 send_signal,wait
//==========================================================================
uint_8 i2c_read1(uint_8 I2C_No,uint_8 DeviceAddr, uint_8 DataLocation, \
                                                   uint_8 * Data)
{
        //获取 i2c 模块基址
```

```
        I2C_MemMapPtr num = i2c_get_base_address(I2C_No);
        BSET(I2C_C1_TX_SHIFT,I2C_C1_REG(num));      //TX = 1,MCU 设置为发送模式
        send_signal('S',I2C_No);                     //发送开始信号
        I2C_D_REG(num) = DeviceAddr & 0xfe;          //发送设备地址,并通知从机接收数据
        if (wait('T',I2C_No))                        //等待一个字节数据传送完成
        {
            return 1;                                //没有传送成功,读一个字节失败
        }
        if (wait('A',I2C_No))                        //等待从机应答信号
        {
            return 1;                                //没有等到应答信号,读一个字节失败
        }
        I2C_D_REG(num) = DataLocation;               //发送数据在从机接收数组中的位置
        if (wait('T',I2C_No))                        //等待一个字节数据传送完成
        {
            return 1;                                //没有传送成功,读一个字节失败
        }
        if (wait('A',I2C_No))                        //等待从机应答信号
        {
            return 1;                                //没有等到应答信号,读一个字节失败
        }
        // 当 MCU 在主机模式下,向该位写 1 将产生一个重新开始信号
        BSET(I2C_C1_RSTA_SHIFT,I2C_C1_REG(num));
        I2C_D_REG(num) = DeviceAddr | 0x01;          //通知从机改为发送数据
        if (wait('T',I2C_No))                        //等待一个字节数据传送完成
        {
            return 1;                                //没有传送成功,读一个字节失败
        }
        if (wait('A',I2C_No))                        //等待从机应答信号
        {
            return 1;                                //没有等到应答信号,读一个字节失败
        }
        BCLR(I2C_C1_RSTA_SHIFT,I2C_C1_REG(num));//TX = 0,MCU 设置为接收模式
        *Data = I2C_D_REG(num);                      //读出 IIC1D,准备接收数据
        if (wait('T',I2C_No))                        //等待一个字节数据传送完成
        {
            return 1;                                //没有传送成功,读一个字节失败
        }
        send_signal('O',I2C_No);                     //发送停止信号
        *Data = I2C_D_REG(num);                      //读出接收到的一个数据
            return 0;                                //正确接收到一个字节数据
}

//==========================================================================
//函数名称: i2c_write1
//功能概要: 向从机写一个字节数据
//参数说明: I2C_No: 模块号,其取值为 0,1
//          DeviceAddr: 设备地址范围 1~255
//          DataLocation: 数据在从机接收数组中的位置范围 0~255
//          Data: 要发给从机的一个字节数据
```

```
//函数返回：为 0,成功写一个字节;为 1,写一个字节失败
//函数说明：内部调用 send_signal,wait
//===========================================================

uint_8 i2c_write1(uint_8 I2C_No,uint_8 DeviceAddr, uint_8 DataLocation, \
                                                          uint_8 Data)
{
    //获取 i2c 模块基址
    I2C_MemMapPtr num = i2c_get_base_address(I2C_No);
    BSET(I2C_C1_TX_SHIFT,I2C_C1_REG(num)); //TX = 1,MCU 设置为发送模式
    send_signal('S',I2C_No);                   //发送开始信号
    I2C_D_REG(num) = DeviceAddr & 0xfe;        //发送设备地址,并通知从机接收数据
    if (wait('T',I2C_No))                      //等待一个字节数据传送完成
        return 1;                              //没有传送成功,写一个字节失败
    if (wait('A',I2C_No))                      //等待从机应答信号
        return 1;                              //没有等到应答信号,写一个字节失败
    I2C_D_REG(num) = DataLocation;             //发送数据在从机接收数组中的位置
    if (wait('T',I2C_No))                      //等待一个字节数据传送完成
        return 1;                              //没有传送成功,写一个字节失败
    if (wait('A',I2C_No))                      //等待从机应答信号
        return 1;                              //没有等到应答信号,写一个字节失败
    I2C_D_REG(num) = Data;                     //写数据
    if (wait('T',I2C_No))                      //等待一个字节数据传送完成
        return 1;                              //没有传送成功,写一个字节失败
    if (wait('A',I2C_No))                      //等待从机应答信号
        return 1;                              //没有等到应答信号,写一个字节失败
    send_signal('O',I2C_No);                   //发送停止信号
        return 0;
}
//===========================================================
//函数名称：i2c_readn
//功能概要：从从机读 N 个字节数据
//参数说明：I2C_No：模块号,其取值为 0,1
//          DeviceAddr：设备地址范围 1～255
//          DataLocation：访问地址范围 0～255
//          Data：读出数据的缓冲区
//          N：从从机读的字节个数
//函数返回：为 0,成功读 N 个字节;为 1,读 N 个字节失败
//函数说明：内部调用 i2c_read1
//===========================================================
uint_8 i2c_readn(uint_8 I2C_No,uint_8 DeviceAddr,uint_8 DataLocation, \
                                              uint_8 Data[], uint_8 N)
{
    uint_16 i, j;
    for (i = 0;i < N;i++)
    {
        for(j = 0;j <15 * 40;j++);         //最小延时(发送的每个字节之间要有时间间隔)
        if (i2c_read1(I2C_No,DeviceAddr, DataLocation + i, &Data[i]))
            return 1;                      //其中一个字节没有接收到,返回失败标志:1
    }
```

```
    if (i >= N)
        return 0;                          //成功接收 N 个数据,返回成功标志:0
}


//======================================================================
//函数名称: i2c_writen
//功能概要: 向从机写 N 个字节数据
//参数说明: I2C_No: 模块号,其取值为 0,1
//          DeviceAddr: 设备地址范围 1~255
//          DataLocation: 数据在从机接收数组中的位置范围 0~255
//          Data: 要写入的数据的首地址
//          N: 从从机读的字节个数
//函数返回: 为 0,成功写 N 个字节;为 1,写 N 个字节失败
//函数说明: 内部调用 i2c_write1
//======================================================================
uint_8 i2c_writen(uint_8 I2C_No,uint_8 DeviceAddr, uint_8 DataLocation,\
                                                  uint_8 Data[], uint_8 N)
{
    uint_16 i, j;
    for (i = 0;i < N;i++)
    {
        for(j = 0;j < 15 * 40;j++);        //最小延时(发送的每个字节之间要有时间间隔)
        if (i2c_write1(I2C_No,DeviceAddr, DataLocation + i, Data[i]))
            return 1;                      //其中一个字节没有发送出去,返回失败标志:1
    }
    if (i >= N)
        return 0;                          //成功发送 N 个数据,返回成功标志:0
}
//======================================================================
//函数名称: i2c_re_enable_int.
//功能说明: 打开 i2c 的 IRQ 中断
//函数参数: i2cNO:i2c 模块号,其取值为 0,1
//函数返回: 无
//======================================================================
void i2c_enable_re_int(uint_8 I2C_No)
{
     enable_irq (I2C_No+8);
}

//======================================================================
//函数名称: i2c_re_disable_int.
//功能说明: 关闭 i2c 的 IRQ 中断
//函数参数: i2cNO:i2c 模块号,其取值为 0,1
//函数返回: 无
//======================================================================
void i2c_disable_re_int(uint_8 I2C_No)
{
     disable_irq (I2C_No+8);
}
```

```
//==========================================================
//函数名称: i2c_get_base_address
//功能概要: 获取 i2c 模块的基址
//参数说明: i2cNO:i2c 模块号,其取值为 0,1
//函数返回: i2c 模块的基址值
//==========================================================
I2C_MemMapPtr i2c_get_base_address(uint_8 I2C_No)
{
        switch(I2C_No)
        {
            case 0:
                    return I2C0_BASE_PTR;
                      break;
            case 1:
                    return I2C1_BASE_PTR;
                       break;
    }
}
```

注：主函数及测试实例见网上教学资源。

11.3　触摸感应接口 TSI 模块

触摸感应输入(Touch Sensing Input,TSI)模块具有高灵敏和强鲁棒性的电容触摸感应检测能力。TSI 模块可在低功耗模式下运行,能以一个触摸事件唤醒 CPU,能够实现键盘触摸、旋转和滑动。本节给出触摸感应接口 TSI 的通用基础知识、TSI 驱动构件及使用方法、TSI 模块的编程结构及驱动构件的设计方法。

11.3.1　触摸感应接口 TSI 的通用基础知识

使用 TSI 作为输入的电气设备,不需要操作人员直接接触电路即可感应到用户的操作,因此,TSI 模块可用于进行基于接近感应的人机交互设备的设计,实现操作人员与电气设备的隔离,这在丰富操作方式的基础上,也提供了更高的安全性能。同时,避免了对设备的直接操作,也使得设备损坏的概率降低,从而减少了维护成本。常见的基于 TSI 模块设计的输入设备应用有:触摸键盘、触摸显示屏等。

1. TSI 触摸感应原理

根据电子学的知识可知,未接地的电极与地之间存在电容。人体可以当作是一个接地面(虚地)如图 11-14 所示,当有人体接近电极板时,等效地增大了电极与地之间的有效面积,使电极板电容值增大。TSI 模块的内部机制可以实现对电极电容值的检测,并且可以设定触发检测事件的阈值。当检测到电容值大于

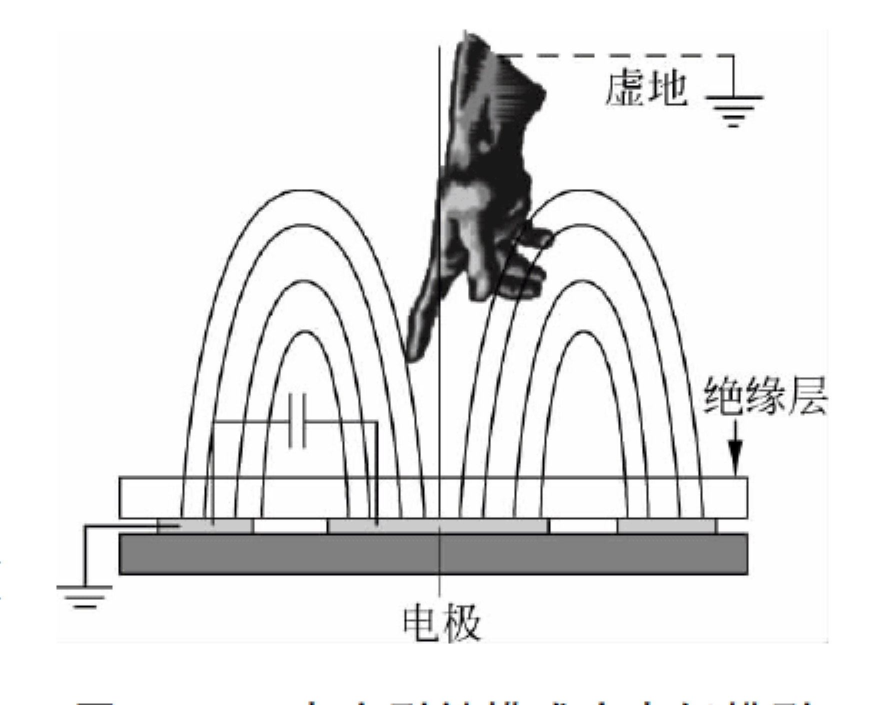


图 11-14　电容型触摸感应电极模型

设定阈值时，TSI的触发标志位将置位，并可激活发出中断请求，从而实现了触摸感应事件的响应。

2. TSI模块测量电容基本原理

TSI模块内部具有两个电流源对外接电极进行充放电，在电极板上产生三角波信号，如图11-15所示。电极上三角波信号的频率随电极电容变化而变化，当电极电容增大时，三角波信号的频率减小，周期变大。TSI模块以一个内部振荡器产生的时钟信号为参考节拍，对电极上的三角波电压信号的周期进行测量计数，当三角波电压信号周期增大时，对应计数值也会增大，如图11-16所示。扫描的计数结果保存在TSI模块的计数寄存器(TSICNT)中，可以通过程序进行访问。

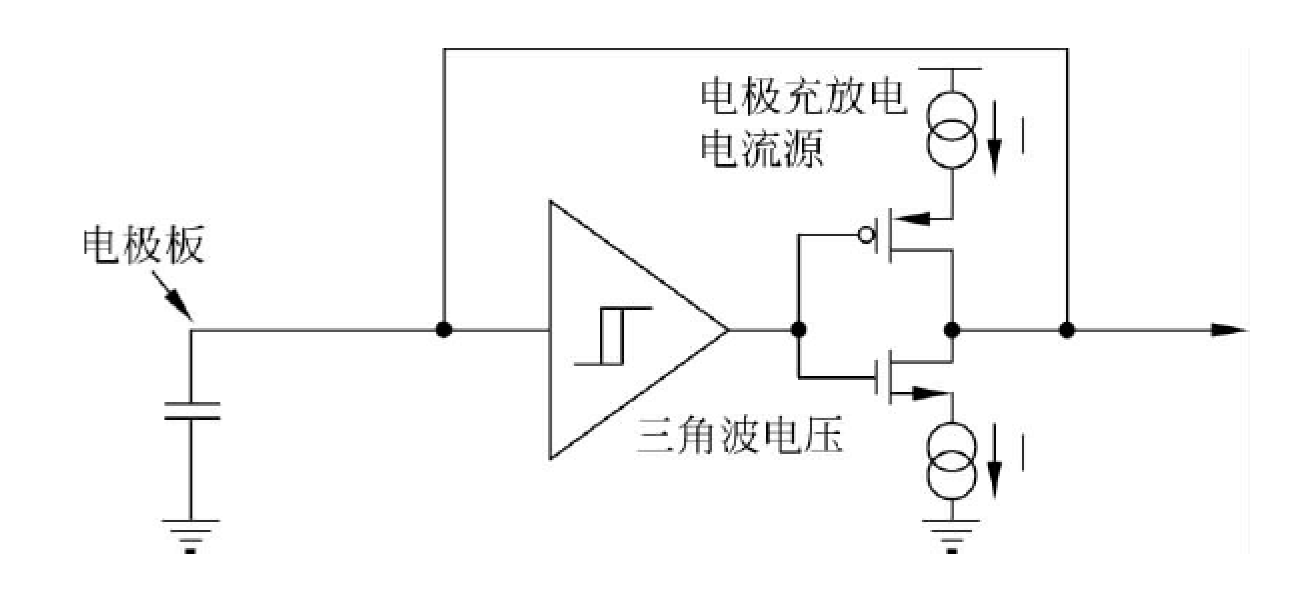


图11-15　TSI电流源对电极进行充放电

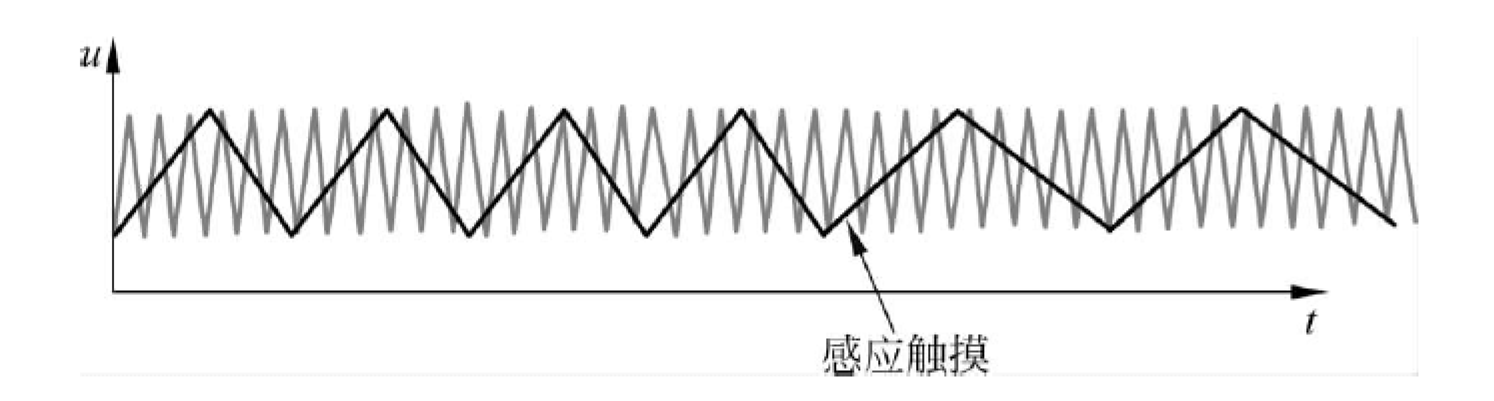


图11-16　对感应信号频率进行计数

TSI模块将每次获得的计数值与存放在阈值寄存器(THRESHLDn)中预设的阈值比较，若超出设定阈值的范围，则会置TSI扫描计数值超出范围标志位(OUTRGF)。此时，若使能TSI中断，则进入TSI中断服务程序响应TSI触发事件。

3. 关于外接电极及电气参数的说明

电极是一块表面覆有绝缘材料的导电板，其与TSI模块的基本连接方式如图11-17所示。这是最简单的TSI电极连接方式，MCU的TSI引脚与电极板之间串联了一个限流电阻，防止电极上与MCU之间的电流过大损害MCU。在实际使用时，还需要考虑实际情况设计电路。一般情况下，电极表面覆盖的绝缘材料(玻璃、绝缘涂层等)厚度约为电极直径的10%，若电极直径为1cm，则合适的感应距离为1mm。

4. 电容测量

电极引脚电容测量使用双晶振的方法。TSI电极晶振有它自己的频率，该频率取决于外部电极电容和TSI模块配置。在自己晶振频率达到可配置的分频器里面的值之后，TSI电极晶振信号进入模数计数器的输入通道。使用TSI参考晶振可以衡量外部电极晶振的时间。电极电容的测量值直接和这个时间成比例。

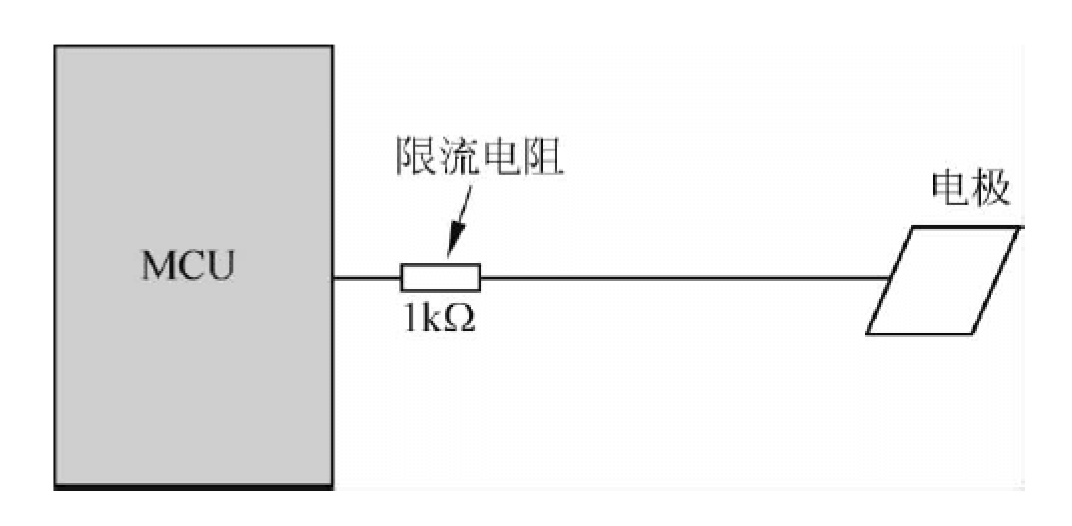


图 11-17　基本的电极接入方法

1）TSI 电极振荡器

TSI 电极振荡器的电路是一个可配置的直流电源对外部电极电容进行充电和放电。一个缓冲滞环规定了振荡器的三角波电压，而三角波电压规定了充放电压的上下限。应用于平板电容的电流源大小由 SCANC[EXTCHRG]位进行选择。

振荡器频率由以下方程给出：

$$F_{\text{elec}} = \frac{I}{2 \times C_{\text{elec}} \times \Delta V}$$

其中，I 为直流电流大小，C_{elec}为电极电容，ΔV 为滞环三角波电压差值。

2）电极振荡器和计数器控制

TSI 振荡器频率信号首先要经过由 GENCS[PS]定义的分频器，之后进入计数器。GENCS[NSCN]位为每个外部电极定义了扫描的次数。

引脚电容采样时间由模数计数器的值从 0 计算到它的最大值得出，最大值由 NSCN 位定义。电极采样时间可以用以下方程式表达：

$$T_{\text{cap_samp}} = \frac{\text{PS} \times \text{NSCN}}{F_{\text{elec}}} = \frac{2 \times \text{PS} \times \text{NSCN} \times C_{\text{elec}} \times \Delta V}{I}$$

其中，PS 为分频器的值，NSCN 为扫描次数，I 为直流电流，C_{elec}为电极电容，ΔV 为滞环三角波电压差值。

3）TSI 参考振荡器

TSI 参考振荡器有着和 TSI 电极振荡器相同的架构。电极振荡器使用外部的电容器，而 TSI 参考振荡器使用可编程的内部参考电容器。电流源由 SCANC[REFCHRG]位定义。

参考振荡器频率由以下方程给出：

$$F_{\text{ref_osc}} = \frac{I_{\text{ref}}}{2 \times C_{\text{ref}} \times \Delta V}$$

其中，C_{ref}为内部参考电容器的电容，I_{ref}为参考振荡器电流源，ΔV 为滞环三角波电压差值。

5. TSI 测量结果

在采样期间，电容测量结果由 TSI 参考振荡器周期值来定义，储存到 TSICHnCNT 寄存器中。

$$\text{TSICHnCNT} = T_{\text{cap_samp}} \times F_{\text{ref_osc}}$$

由上述方程可以得到以下方程：

$$\text{TSICHnCNT} = \frac{I_{\text{ref}} \times \text{PS} \times \text{NSCN}}{C_{\text{ref}} \times I_{\text{ref}}} \times C_{\text{elec}}$$

11.3.2　TSI 驱动构件头文件及使用方法

KL25 的 TSI 模块提供的是一种电容感应输入接口，包括 16 个 TSI 引脚，当有感应物接近与 TSI 引脚相连的电极时，通过 TSI 模块可以检测电极板的电容值的变化，为判断触摸感应提供依据。

1. TSI 引脚

MKL25Z128VLK4 芯片只有一个 TSI 模块，标记为 TSI0。它的通道并不是固定在哪几个引脚上，而是可以通过引脚配置寄存器配置。根据附录 A(MKL25Z128VLK4 引脚功能分配)，可以配置为串口的引脚及 SD-FSL-KL25-EVB 实际使用的引脚见表 11-15。

表 11-15　KL25 的 TSI 引脚及 SD-FSL-KL25-EVB 使用的引脚

引脚号	引脚名	ALT0	ALT1	ALT2	ALT3	ALT4	SD-FSL-KL25-EVB
26	PTA0	TSI0_CH1	PTA0				
27	PTA1	TSI0_CH2	PTA1	UART1_RX	TPM2_CH0		
28	PTA2	TSI0_CH3	PTA2	UART1_TX	TPM2_CH1		
29	PTA3	TSI0_CH4	PTA3	I2C0_SCL	TPM1_CH0		
30	PTA4	TSI0_CH5	PTA4	I2C0_SDA	TPM1_CH1		TSI0_CH5
43	PTB0	ADC0_SE8/TSI0_CH0	PTB0/LLWU_P5	I2C0_SCL	TPM1_CH0		
44	PTB1	ADC0_SE9/TSI0_CH6	PTB1	I2C0_SDA	TPM1_CH1		
45	PTB2	ADC0_SE12/TSI0_CH7	PTB2	I2C0_SCL	TPM1_CH0		
46	PTB3	ADC0_SE13/TSI0_CH8	PTB3	I2C0_SDA	TPM1_CH1		
51	PTB16	TSI0_CH9	PTB16		UART0_RX	TPM_CLKIN0	
52	PTB17	TSI0_CH10	PTB17		UART0_TX	TPM_CLKIN1	
53	PTB18	TSI0_CH11	PTB18		TPM2_CH0		
54	PTB19	TSI0_CH12	PTB19		TPM2_CH1		
55	PTC0	ADC0_SE14/TSI0_CH13	PTC0		ETRG_IN		
56	PTC1	ADC0_SE15/TSI0_CH14	PTC1	I2C1_SCL		TPM0_CH0	
57	PTC2	ADC0_SE11/TSI0_CH15	PTC2	I2C1_SDA		TPM0_CH1	

2. TSI 驱动构件基本要点分析

TSI 驱动构件由头文件 tsi.h 及源代码文件 tsi.c 组成，放入 tsi 文件夹中，供应用程序开发调用。

TSI 具有初始化、获取返回值和设置 TSI 阈值三种基本操作。按照构件的思想，可将它们封装成三个独立的功能函数。TSI 初始化函数 tsi_init 主要完成对 TSI 模块工作的参数设定，包括工作时钟、工作方式、电气参数、引脚门控使能及模块使能等。默认未开启中断，若要使用中断触发模式，可以使用 TSI_ENABLE 宏定义进行一些中断触发设置的操作。TSI 获取返回值函数 tsi_get_value16 主要是启动一次 TSI 扫描，获取 TSI 通道的计数值，将结果保存数返回。设置 TSI 阈值函数 tsi_set_threshold 主要是设定 TSI 通道的触发阈值，设定的触发阈值包括阈值下限和阈值上限，当让 TSI 模块自动进行超出范围判断时，若 TSI 通道计数值超出设定阈值的上下限，则 TSI 模块认为 TSI 引脚上有 TSI 事件触发，将会自动设置 TSI 触发标志位，当设定中断时产生中断服务请求。除此之外，还有使能 TSI 模块函数 tsi_enable_re_int（用来开 TSI 中断）、关闭 TSI 模块函数 tsi_disable_re_int（用来

关 TSi 中断)和开启一次软件扫描函数 tsi_softsearch。

通过以上分析,可以设计 TSI 构件的几个基本功能函数。

(1) TSI 初始化:void tsi_init(uint_8 chnlID);

(2) 获取返回值:uint_16 tsi_get_value16();

(3) 设置 TSI 阈值:void tsi_set_threshold(uint_16 low, uint_16 high);

(4) 使能 TSI 模块:void tsi_enable_re_int();

(5) 关闭 TSI 模块:void tsi_disable_re_int();

(6) 开启一次软件扫描:void tsi_softsearch();

3. TSI 驱动构件头文件

```
//===========================================================================
//文件名称: tsi.h
//功能概要: KL25 tsi 底层驱动程序头文件
//版权所有: 苏州大学恩智浦嵌入式中心(sumcu.suda.edu.cn)
//版本更新: 2012-11-25   V1.0    初始版本
//          2015-03-18 v2.1
//          2016-04-13 v2.2
//===========================================================================

#ifndef TSI_H                                   //防止重复定义(开头)
#define TSI_H
//1 头文件
#include "common.h"                             //包含公共要素头文件
//2 宏定义

//3 函数声明
//===========================================================================
//函数名称: tsi_init
//功能概要: 初始化 TSI 模块,KL25 只有一个 TSI 模块
//参数说明: chnlIDs: 8 位无符号数,TSI 模块所使用的通道号,其取值为 0~15
//函数返回: 无
//===========================================================================
void tsi_init(uint_8 chnlID);

//===========================================================================
//函数名称: tsi_get_value16
//功能概要: 获取 TSI 通道的计数值
//参数说明: 无
//函数返回: 获取 TSI 通道的计数值
//===========================================================================
uint_16 tsi_get_value16();

//===========================================================================
//函数名称: tsi_set_threshold1
//功能概要: 设定指定通道的阈值
//参数说明: low: 设定阈值下限,取值范围为 0~65 535
```

```
//          high: 设定阈值上限,取值范围为0～65 535
//函数返回: 无
//===========================================
void tsi_set_threshold(uint_16 low, uint_16 high);

//===========================================
//函数名称: tsi_enable_re_int
//功能概要: 开 TSI 中断,关闭软件触发扫描,开中断控制器 IRQ 中断
//参数说明: 无
//函数返回: 无
//===========================================
void tsi_enable_re_int();
//===========================================
//函数名称: tsi_disable_re_int
//参数说明: 无
//函数返回: 无
//功能概要: 关 TSI 中断,开软件触发扫描,关中断控制器 IRQ 中断
//===========================================
void tsi_disable_re_int();

//===========================================
//函数名称: tsi_softsearch
//功能概要: 开启一次软件扫描
//参数说明: 无
//函数返回: 无
//===========================================
void tsi_softsearch();

#endif  //防止重复定义(结尾)
```

4. TSI 驱动构件使用方法

在 TSI 驱动构件的头文件(tsi.h)中包含的内容有初始化 TSI 模块(tsi_init)、获取 TSI 通道的计数值(tsi_get_value16)、设定 TSI 通道的阈值(tsi_set_threshold)、开 TSI 中断(tsi_enable_re_int)、关 TSI 中断(tsi_disable_re_int)、开启一次软件扫描(tsi_softsearch)。

下面介绍构件的使用方法,举例如下。

(1) 在主函数 main 中,首先定义 TSI 模块所使用的通道号并赋值,然后调用初始化函数,传入通道号。

```
uint_8 chnlID=5;            //TSI 通道测试选择通道 5
tsi_init(chnlID);           //初始化 TSI
```

(2) 在头文件 include 中定义两个宏分别表示通道阈值下限和上限,然后调用设定通道阈值的函数,设置指定通道的阈值。其中,传入的第一个参数为下限,第二个为上限。

```
#define TSI_TSHD_VALUE_HIGH     0x010c
#define TSI_TSHD_VALUE_LOW      0x004F
tsi_set_threshold(TSI_TSHD_VALUE_LOW, TSI_TSHD_VALUE_HIGH);
```

(3) 调用 TSI 模块中断使能函数，开 TSI 中断。

```
tsi_enable_re_int();        //开 TSI 中断
```

(4) 开启一次软件扫描。

```
tsi_softsearch();
```

(5) 当获得的通道计数值超出阈值范围时，会产生 TSI 中断。在中断函数中获取计数值，并把它通过串口 1 发送给 PC。

```
uint_16 i;
i = tsi_get_value16();
uart_send1(UART_1, (uint_8)(i)+'0');
```

5. TSI 驱动构件测试实例

测试工程位于网上教学资源中的"..\ KL25-program\ch11-KL25-SPI-I2C-TSI"文件夹中，其功能如下。

(1) 串口通信格式：使用串口 1，波特率 9600，1 位停止位，无校验。

(2) 上电或按复位按钮时，调试串口输出"This is TSI Test!"。

(3) 初始化红灯和蓝灯为暗，然后主循环中蓝灯闪烁，当 TSI 通道计数值超过预定的阈值的上下限时，将产生 TSI 中断，在中断函数中通过串口 1 把溢出值发给 PC，同时将红灯点亮。

(4) PC 向 MCU 发送数据时，MCU 进入串口接收中断，将接收的一个字节直接回发。

11.3.3 TSI 模块的编程结构

KL25 的 TSI 模块共有三个 32 位寄存器，包括一个通用控制和状态寄存器(TSI0_GENCS)，DATA 寄存器(TSI0_DATA)和阈值寄存器(TSI0_TSHD)。通过对这些寄存器的编程，就可以使用 TSI 模块进行电容的测量。

1. 通用控制和状态寄存器

通用控制和状态寄存器(TSI0_GENCS)主要完成 TSI 模块各种各样的控制和状态信息的配置，其结构如表 11-16 所示。所有的位复位之后都为 0。

表 11-16　TSI0_GENCS 结构

数据位	D31	D30、D29	D28	D27～D24	D23～D21	D20、D19	D18～D16
读	OUTRGF	0	ESOR	MODE	REFCHRG	DVOLT	EXTCHRG
写	W1C	—					

数据位	D15～D13	D12～D8	D7	D6	D5	D4	D3	D2	D1	D0
读	PS	NSCN	TSIEN	TSIIEN	STPE	STMS	SCNIP	EOSF	CURSW	0
写							—	W1C		—

D31(OUTRGF)——表示TSI_DATA[TSICNT]中的数据是否超出TSI_THRESHLD中的阈值。若超出,置1。

D30、D29——该位段保留且只读为0。

D28(ESOR)——中断类型选择,1表示扫描结束中断;0表示超过阈值产生中断。

D27～D24(MODE)——TSI模拟模式状态设置位。0000表示TSI在电容检测模式下工作。0100表示TSI为单个阈值噪声检测模式,但频率限制电路禁用。1000表示TSI为单个阈值噪声监测模式,并且频率限制电路使能。1100表示TSI模块开启自动噪声检测模式。

D23～D21(REFCHRG)——表示选择参考振荡器冲放电时的电流值。000表示500nA,001表示1μA,010表示2μA,011表示4μA,100表示8μA,101表示16μA,110表示32μA,111表示64μA。

D20、D19(DVOLT)——表示振荡器充放电电压的峰谷值。00表示Dv=1.03V,Vp=1.33V,Vm=0.30V。01表示Dv=0.73V,Vp=1.18V,Vm=0.45V。10表示Dv=0.43V,Vp=1.03V,Vm=0.60V。11表示Dv=0.29V,Vp=0.95V,Vm=0.67V。其中,Dv是电压峰值与电压谷值之间的差值,Vp是电压峰值,Vm是电压谷值。

D18～D16(EXTCHRG)——用来设置电极振荡器的充放电电流值。000表示500nA,001表示1μA,010表示2μA,011表示4μA,100表示8μA,101表示16μA,110表示32μA,111表示64μA。

D15～D13(PS)——电极振荡器输出频率的预分频值。000表示原频率值,001表示原频率1/2,010表示原频率1/4,011表示原频率1/8,100表示原频率1/16,101表示原频率1/32,110表示原频率1/64,111表示原频率1/128。

D12～D8(NSCN)——这些位表示对每个电极每次扫描的次数,扫描的次数是TSI_GENCS[NSCN]的值加1。

D7(TSIEN)——TSI模块使能位。0表示TSI模块禁用,1表示TSI模块使能。

D6(TSIIEN)——TSI中断使能位。该中断能把CPU在低功耗模式下唤醒。0表示TSI中断禁用,1表示TSI中断使能。

D5(STPE)——0表示TSI在低功耗模式下禁用,1表示TSI能在低功耗模式下运行。

D4(STMS)——0表示软件触发扫描,1表示硬件触发扫描。

D3(SCNIP)——扫描过程标志位。该位为只读位,并且由TSI自动设置。1表示扫描正在进行中,0表示没有扫描在进行中。

D2(EOSF)——扫描结束标志位。0表示扫描没有完成,1表示扫描完成。

D1(CURSW)——该位表示电极振荡器和参考振荡器的电流源是否交换。0表示电流源不交换,1表示电流源交换。

D0——该位保留且只读为0。

2. TSI_DATA寄存器

TSI_DATA寄存器(TSIx_DATA)主要用于TSI通道的设置、计数值的获取以及软件触发开始的设置,其结构如表11-17所示。所有位复位之后都为0。

表 11-17　TSIx_DATA 结构

数据位	D31～D28	D27～D24	D23	D22	D21～D16	D15～D0
读	TSICH	0	DMAEN	0	0	TSICNT
写		—		SWTS	—	—

D31～D28(TSICH)——表示选择 TSI 通道，通道为 TSI_DATA[TSICH]值加 1。

D27～D24——该位段保留且只读为 0。

D23(DMAEN)——DMA 传输允许。该位通常和 TSIIE、ESOR 一起使用，用来生成 DMA 传输请求。0 表示当 TSI 中断使能并且相应的 TSI 事件触发时，产生中断。1 表示当 TSI 中断使能并且相应的 TSI 事件触发时，产生 DMA 传输请求。

D24(SWTS)——软件触发开始。该只写位是一个软件触发位。当 STM 位被清除时，写 1 到该位将产生一次扫描。被扫描的电极通道由 TSICH 位决定。0 表示无效，1 表示扫描一次。

D21～D16——该位段保留且只读为 0。

D15～D0(TSICNT)——TSI 计数值。该只读位记录了累加扫描所得的计数值。

3. 阈值寄存器

阈值寄存器(TSIx_TSHD)主要用于阈值上下限的设置。

阈值寄存器(TSIx_TSHD)的前 16 位表示阈值下限 D15～D0(THRESL)，后 16 位表示阈值上限 D31～D16(THRESH)。复位之后都为 0。

11.3.4　TSI 驱动构件的设计

1. TSI 基本编程步骤

实现 TSI 的电容测量主要涉及以下几个寄存器：通用控制和状态寄存器(TSI0_GENCS)、DATA 寄存器(TSI0_DATA)、阈值寄存器(TSI0_TSHD)。其中，通用控制和状态寄存器 TSI0_GENCS 用于对 TSI 中断使能，TSI 模块使能，中断类型选择，参考振荡器充放电时的电流值，振荡器充放电电压的峰值，电极振荡器的充放电电流值，电极每次的扫描次数，阈值的超出，触发扫描的方式等的配置；DATA 寄存器 TSI0_DATA 用于 TSI 通道选择和软件触发开始的设置；阈值寄存器 TSI0_TSHD 用于 TSI 阈值上下限的设置。基本编程步骤如下。

(1) 打开 TSI 模块时钟源 SIM_SCG5_TSI。

(2) 引脚复用为 TSI 功能。选择 PTA14 引脚配置通道 5 使能。

(3) 配置通用控制和状态寄存器 TSI0_GENCS，使能 TSI 模块(GENCS [TSIEN]=1)，TSI 中断使能(GENCS [TSIIEN]=1)，以及中断类型选择，参考振荡器充放电时的电流值，振荡器充放电电压的峰值，电极振荡器的充放电电流值，电极每次的扫描次数，阈值的超出，触发扫描的方式等的设置。

(4) 配置 DATA 寄存器 TSI0_DATA，选择通道 5。

(5) 配置阈值寄存器 TSI0_TSHD，设定指定通道的阈值。

2. TSI 驱动构件源码

```
//==========================================================================
//文件名称：tsi.c
//功能概要：KL25 tsi 底层驱动程序文件
//版权所有：苏州大学恩智浦嵌入式中心(sumcu.suda.edu.cn)
//版本更新：2012-11-25   V1.0    初始版本
//          2013-05-05 v2.1
//          2016-04-12 v2.2
//==========================================================================
#include "tsi.h"
//==========================================================================
//函数名称：tsi_init
//功能概要：初始化 TSI 模块,KL25 只有一个 TSI 模块
//参数说明：chnlIDs：8 位无符号数,TSI 模块所使用的通道号,其取值为 0～15

//函数返回：无
//==========================================================================
void tsi_init(uint_8 chnlID)
{
    //开启 TSI 时钟
    BSET(SIM_SCGC5_TSI_SHIFT,SIM_SCGC5);
    BSET(SIM_SCGC5_PORTA_SHIFT,SIM_SCGC5);

    //通道号：0=PTB0 脚,1=PTA0 脚,2=PTA1 脚,3=PTA2 脚,4=PTA3 脚,5=PTA4 脚,
    //         6=PTB1 脚,7=PTB2 脚,8=PTB3 脚,9=PTB16 脚,10=PTB17 脚,
    //         11=PTB18 脚,12=PTB19 脚,13=PTC0 脚,14=PTC1 脚,15=PTC2 脚
    //除了 1、2、3、4、5 以外其他引脚的默认功能即为 TSI 通道
    switch(chnlID)    //chnlID 的取值为 0～15
    {
    case 1:
        PORTA_PCR0 = PORT_PCR_MUX(0);             //通道 1 使能
    case 2:
        PORTA_PCR1 = PORT_PCR_MUX(0);             //通道 2 使能
    case 3:
        PORTA_PCR2 = PORT_PCR_MUX(0);             //通道 3 使能
    case 4:
        PORTA_PCR3 = PORT_PCR_MUX(0);             //通道 4 使能
    case 5:
        PORTA_PCR4 = PORT_PCR_MUX(0);             //通道 5 使能
    }

    BSET(TSI_GENCS_TSIIEN_SHIFT,TSI0_GENCS);      //TSI 中断使能
    BSET(TSI_GENCS_STPE_SHIFT,TSI0_GENCS);        //TSI 在低功耗模式下运行
    //寄存器 TSI0_GENCS 中 REFCHRG 位置位 4,即参考振荡器充放电电流为 8μA
    TSI0_GENCS |= (TSI_GENCS_REFCHRG(4)
        | TSI_GENCS_DVOLT(0)     //寄存器 TSI0_GENCS 中 DVOLT 位为 00 表示峰值电压
                    //Vp=1.33V,谷值电压 Vm=0.30V,峰值谷值之差 Dv=1.03V
            | TSI_GENCS_EXTCHRG(6)                //电极振荡器充放电电流值 32μA
        | TSI_GENCS_PS(2)                         //电极振荡器 4 分频
        | TSI_GENCS_NSCN(11)                      //每个电极扫描 4 次
    );
```

```
    BCLR(TSI_GENCS_ESOR_SHIFT,TSI0_GENCS);          //设置超过阈值产生中断
    BCLR(TSI_GENCS_STM_SHIFT,TSI0_GENCS);           //软件触发扫描
    //清越值标志位和扫描完成位
    //超出阈值置位,EOSF 为 1 设置成扫描完成状态
    BSET(TSI_GENCS_OUTRGF_SHIFT,TSI0_GENCS);
    BSET(TSI_GENCS_EOSF_SHIFT,TSI0_GENCS);
    //选择通道
    TSI0_DATA |= (TSI_DATA_TSICH(chnlID));
    //TSI 模块使能
    BSET(TSI_GENCS_TSIEN_SHIFT,TSI0_GENCS);
}


//==========================================
//函数名称: tsi_get_value16
//功能概要: 获取 TSI 通道的计数值
//参数说明: 无
//函数返回: 获取 TSI 通道的计数值
//==========================================
uint_16 tsi_get_value16()
{

    uint_16 value;
    BCLR(TSI_GENCS_TSIIEN_SHIFT,TSI0_GENCS);        //关 TSI 中断
    BSET(TSI_DATA_SWTS_SHIFT,TSI0_DATA);
    //TSI0_DATA |= TSI_DATA_SWTS_MASK;              //扫描一次选定的通道
    while(!(TSI0_GENCS & TSI_GENCS_EOSF_MASK));     //等待扫描完成
    BSET(TSI_GENCS_EOSF_SHIFT,TSI0_GENCS);          //写 1 清 0 扫描结束标志位
    //TSI0_GENCS |= TSI_GENCS_EOSF_MASK;            //写 1 清 0 扫描结束标志位
    value = (TSI0_DATA & TSI_DATA_TSICNT_MASK);     //读取计数寄存器中的值
    BSET(TSI_GENCS_OUTRGF_SHIFT,TSI0_GENCS);        //写 1 清 0 超值标志位
    //TSI0_GENCS |= TSI_GENCS_OUTRGF_MASK;          //写 1 清 0 超值标志位
    BSET(TSI_GENCS_EOSF_SHIFT,TSI0_GENCS);          //清扫描结束标志位
    //TSI0_GENCS |= TSI_GENCS_EOSF_MASK;            //清扫描结束标志位
    BSET(TSI_GENCS_TSIIEN_SHIFT,TSI0_GENCS);        //开 TSI 中断
    return value;
}


//==========================================
//函数名称: tsi_set_threshold1
//功能概要: 设定指定通道的阈值
//参数说明: low: 设定阈值下限,取值范围为 0~65 535
//          high: 设定阈值上限,取值范围为 0~65 535
//函数返回: 无
//==========================================
void tsi_set_threshold(uint_16 low, uint_16 high)
{
    uint_32 thresholdValue;
    //高 16 位为上限,低 16 位为下限
```

```
    thresholdValue = high;
    thresholdValue = (thresholdValue<<16)|low;
    TSI0_TSHD = thresholdValue;
}


//==========================================================================
//函数名称: tsi_enable_re_int
//功能概要: 开 TSI 中断,关闭软件触发扫描,开中断控制器 IRQ 中断
//参数说明: 无
//函数返回: 无
//==========================================================================
void tsi_enable_re_int()
{
    //开 TSI 中断,关闭软件触发扫描
    BSET(TSI_GENCS_TSIIEN_SHIFT,TSI0_GENCS);
    BSET(TSI_GENCS_STM_SHIFT,TSI0_GENCS);
    enable_irq(26);                                  //开中断控制器 IRQ 中断
}

//==========================================================================
//函数名称: tsi_disable_re_int
//参数说明: 无
//函数返回: 无
//功能概要: 关 TSI 中断,开软件触发扫描,关中断控制器 IRQ 中断
//==========================================================================
void tsi_disable_re_int()
{
    //关 TSI 中断,开软件触发扫描
    BCLR(TSI_GENCS_TSIIEN_SHIFT,TSI0_GENCS);
    BCLR(TSI_GENCS_STM_SHIFT,TSI0_GENCS);
    //禁止中断控制器 IRQ 中断
    disable_irq(26);
}

//==========================================================================
//函数名称: tsi_softsearch
//功能概要: 开启一次软件扫描
//参数说明: 无
//函数返回: 无
//==========================================================================
void tsi_softsearch()
{
    BCLR(TSI_GENCS_STM_SHIFT,TSI0_GENCS);
    //TSI0_GENCS &= ~TSI_GENCS_STM_MASK;    //开启软件触发
    BSET(TSI_DATA_SWTS_SHIFT,TSI0_DATA);
    //TSI0_DATA |= TSI_DATA_SWTS_MASK;      //开始一次软件扫描
}
```

小　结

本章在主要阐述 SPI、I2C、TSI 的工作原理，给出它们的编程步骤和方法，并给出了工程样例。

(1) SPI 一般使用 4 条线：串行时钟线 SCK、主机输入/从机输出数据线 MISO、主机输出/从机输入数据线 MOSI 和从机选择线。SPI 通信过程中需要掌握的基本概念有主机、从机、同步、双工通信、时钟极性、时钟相位和波特率等。11.1 节给出了 SPI 驱动构件的 SPI.h 和 SPI.c 文件，在构件中包括模块初始化(SPI_init)、发送数据流(SPI_sendstring)、接收数据流(SPI_receiveN)、启动 SPI 接收中断(SPI_re_enable_int)。

(2) I2C 总线主要用于同一电路板内各集成电路模块之间的连接，采用双向 2 线制(SDA、SCL)串行数据传输方式。在 I2C 总线上，各 IC 除了个别中断引线外，相互之间没有其他连线，用户常用的 IC 基本上与系统电路无关，故极易形成用户自己的标准化、模块化设计。11.2 节给出了 I2C 驱动构件的 i2c.h 和 i2c.c 文件，构件中包括模块初始化(i2c_init)、读一字节(i2c_read1)、写一字节(i2c_write1)等基本操作，并添加了读多字节(i2c_readn)和写多字节(i2c_writen)等常用操作函数。

(3) KL25 的 TSI 是一种电容感应接近传感器，当有感应物接近与 TSI 引脚相连的电极时，通过 TSI 模块可以检测电极板的电容值的变化，为判断触摸感应提供依据。11.3 节给出了 TSI 驱动构件的 tsi.h 和 tsi.c 文件，构件中包含模块初始化(tsi_init)、获取所有通道采样值(tsi_get_value16)、设定单通道触发阈值(tsi_set_threshold)和设定所有通道触发阈值(tsi_set_threshold16)等基本操作函数。

习　题

1. 简述同步通信与异步通信的联系与区别。
2. 简述 SPI 总线的时钟同步过程。
3. 简述 I2C 总线的数据传输过程。
4. 简述 KL25 芯片的 I2C 主机从从机读一个字节数据的过程。
5. 从从机的接入、时钟控制、数据传输速度、是否可以实现多主控、作用领域等方面比较 SPI 和 I2C。
6. 简述 KL25 的 TSI 模块的工作原理。

第 12 章　USB 编程

本章导读：本章是全书的重点和难点之一。主要阐述了 USB 通信接口的优点、工作原理和编程方法。主要内容有：①介绍了 USB 协议基本概念、历史和发展，以及 USB 的基本知识要素；②重点阐述了 USB 通信协议，着重描述了 USB 设备上电的枚举过程；③介绍了 KL25 芯片的 USB 模块的基本特征和硬件连接电路；④介绍了 PC 方 USB 设备驱动程序的选择和基本原理，并提供 PC 的 USB 程序开发方法和测试实例；⑤阐述了 USB 模块基本编程要点和构件设计方法；同时给出了测试方法和测试实例。

12.1　USB 应用开发基础知识

通用串行总线（Universal Serial Bus，USB）是 2000 年以来普遍使用的连接外围设备和计算机的一种新型串行总线标准。与传统计算机接口相比，它克服了对硬件资源独占，限制对计算机资源扩充的缺点，并以较高的数据传输速率和即插即用等优势，逐步发展成为计算机与外设的标准连接方案。现在不但常用的计算机外设如鼠标、键盘、打印机、扫描仪、数码相机、U 盘、移动硬盘等使用 USB 接口，就连数据采集、信息家电、网络产品等领域也越来越多地使用 USB 接口。到 2009 年为止，全球大约已有二十多亿的 USB 设备。图 12-1 给出了通用 USB 标志和 PC 上的 USB 口。

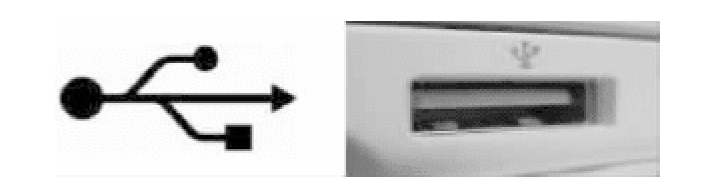


图 12-1　USB 标志和 PC 的 USB 接口

USB 接口之所以被广泛应用，主要与 USB 的如下特点密切相关。

(1) 支持即插即用（Plug-and-Play）。所谓即插即用，包括两方面的内容，一方面是热插拔，即在不需要重启计算机或关闭外设的条件下，便可以实现外设与计算机的连接和断开，而不会损坏计算机和设备；另一方面是可以快速简易安装某硬件设备而无须安装设备驱动程序或重新配置系统。

(2) 可以使用总线电源。USB 总线可以向外提供一定功率的电源，其输出电流的最小值为 100mA，最大为 500mA，输出电压为 5V，适合很多嵌入式系统。USB 协议中定义了完备的电源管理方式，用户可以选择是采用设备自供电还是从 USB 总线上获取电源。

(3) 硬件接插口标准化、小巧化。USB 协议定义了标准的接插口：A 型和 B 型，这样就为种类繁多的 USB 设备提供了统一的硬件接插口。同时 USB 接口和老式的通信接口相比具有明显的体积优势，为计算机外设的小型化发展提供了可能。

(4) 支持多种速度和操作模式。目前 USB 支持三种传输速度：低速 1.5Mb/s、全速 12Mb/s、高速 480Mb/s。2009 年新推出的 USB 3.0 芯片支持超速 5.0Gb/s。同时 USB 还支持 4 种类型的传输模式：块传输、中断传输、同步传输和控制传输，这样可以满足不同外设的功能需求。

12.1.1　USB 的物理特性

1. USB 电缆和接口类型

USB 主机和 USB 设备之间、USB 主机和 Hub 之间以及 USB 设备和 Hub 之间都需要 USB 电缆进行连接。USB 协议规定，对于 USB 高速传输（480Mb/s）和全速传输（12Mb/s），需要使用外壳屏蔽、数据线双绞的 USB 电缆；对于低速传输（1.5Mb/s），USB 电缆不需要使用外壳屏蔽、数据线双绞的 USB 电缆。USB 电缆由 4 根导线组成，分别是一对差分信号线 D＋和 D－、电源线 V_{BUS} 和地线 GND，如图 12-2 所示。一般来说，红色导线表示 V_{BUS}、黑色导线表示 GND、绿色导线表示 D＋、白色导线表示 D－。

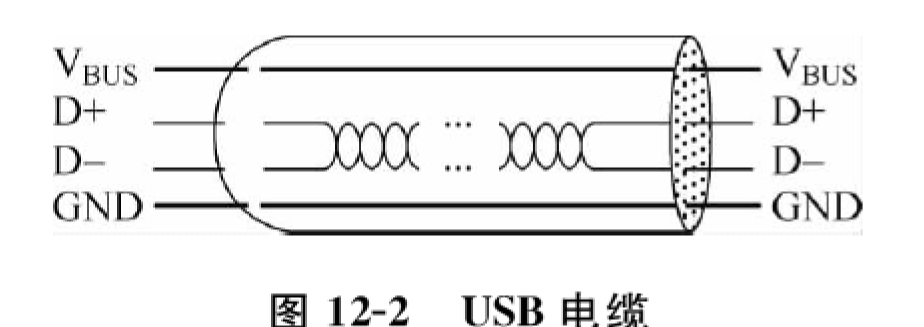


图 12-2　USB 电缆

USB 接口主要有两种：A 型和 B 型，如图 12-3 所示。USB 连接器的插座和插头相互匹配。一般来说，A 型插座用作 USB 主机或 Hub 的下行端口，所以 A 型插头总是指向上行的 USB 主机或 Hub；B 型插座用作 USB 设备或上行端口，所以 B 型插头总是指向下行 USB 设备。USB 连接器的 4 个引脚，分别对应 USB 电缆的 4 根导线。




(a) A型USB接口的插头和插座




(b) B型USB接口的插头和插座

图 12-3　USB 接口类型

2. USB 差分信号

数据在 USB 总线上实际传输时，使用的是 NRZI（反向不归零）编码的差分信息，这种信号有利于保证数据的完整性和消除噪声干扰。

我们知道，传统的传输方式大多使用“正/负信号”技术，即用“正信号”或者“负信号”二进制表达机制。这些信号使用单线传输，用不同的信号电平范围来分别表示 1 和 0，它们之间有一个临界值。如果在数据传输过程中受到中低强度的干扰，高低电平不会突破临界值，那么信号传输可以正常进行。但如果遇到强干扰，高低电平突破临界值，由此将造成数据传输错误。一般来说，总线频率越高，线路间的电磁干扰就越厉害，数据传输失败的发生频率也就越高。因此这种信号表达技术无法应用于高速总线传输，而差分信号技术可以克服该缺点。

差分信号技术最大的特点是必须使用两条线路才能表达一个比特位（bit），用两条线路传输信号的压差作为判断是逻辑 1 还是逻辑 0 的依据，这种做法的优点是具有极强的抗干扰性。如果在数据传输时遭受外界干扰，两条线路对应的电平会出现同样幅度的提升和降低，但两者的电平改变方向和幅度几乎相同，电压差就可以始终保持相对稳定，因此数据的准确性并不会因干扰噪声而有所降低。由于一个比特位需要两条线路，在总线带宽相同的条件下，差分信号技术需要的信号线条数是“正/负信号”技术所需信号线条数的两倍。

3. USB 总线上的状态与设备速度检测

在 USB 协议中，给 USB 总线定义了 4 种状态：SE0、SE1、J 和 K 状态。D＋和 D－数据

线都被拉低时的状态为 SE0 状态；D+和 D−数据线都被拉高时的状态为 SE1 状态(非法状态)。当有设备连接到主机时，D+或 D−被上拉，被上拉的数据线为高电平而另一根数据线为低电平，这种状态为 J 状态(空闲状态)，包(Package)被传输之前和之后，总线就是处于该状态。而 K 状态是指两根数据线的极性与 J 状态下相应的两根数据线的极性相反的状态(若 J 状态为 D+上拉、D−下拉，则 K 状态指 D+下拉、D−上拉)，主机通过 J 状态和 K 状态判断设备是否支持高速传输。

当 USB 主机或 Hub 的下行端口上没有 USB 设备连接时，USB 总线处于 SE0 状态。当有设备连接后，如图 12-4 所示，电流流过由 Hub 的下拉电阻和设备的 D+/D−数据线上的上拉电阻构成分压器。由于下拉电阻的阻值是 15kΩ，上拉电阻的阻值 1.5kΩ，所以在 D+/D−数据线上会出现大小为 $V_{CC}\times15/(15+1.5)$的直流高电平电压。如果主机检测到 D+数据线上为高电压，说明连接上的是高速/全速设备；若检测到 D-数据线上为高电压，说明连接上的是低速设备。



图 12-4　USB 电缆和电阻的连接

高速设备在连接开始时需要以全速速率与主机进行通信，以完成其配置操作。在复位期间(主机将 D+和 D−都拉低，使 USB 总线处于 SE0 状态，并保持至少 10ms。在 2.5μs 后设备将识别复位条件，该复位要与微处理器的上电复位区别开来，它是 USB 协议复位，是为了使设备的 USB 信号开始时是一个已知状态)，支持高速传输的设备会有一个 K 状态(这是由集成在 USB 设备接口芯片的内部软件控制开关完成的)，具有高速能力的 Hub 检测到总线上的该状态并响应一个 K 和 J 的交替状态序列，之后设备将从全速提高到高速。如果 Hub 不响应设备的 K 状态，那么设备就一直用全速通信。

12.1.2　USB 主机与设备的概念与特性

1. USB 主机

USB 主机指的是包含 USB 主控制器，并且能够控制完成与 USB 设备之间数据传输的设备。广义上说，USB 主机包含计算机和具有 USB 主控芯片的设备。USB 的所有数据通信(不论是上行通信还是下行通信)都由 USB 主机发起，所以 USB 主机在这个数据传输过程中占据着主导地位。USB 协议规定，在同一时刻 USB 系统中只允许存在一个 USB 主机，否则会引起控制和传输的混乱。在 PC 或笔记本上，USB 一般作为主机。从开发人员的角度来看，这个 USB 主机可以分为三个不同的功能模块：客户软件、USB 系统软件和 USB 总线接口。

(1) 客户软件。客户软件负责和 USB 设备的功能单元进行通信,以实现其特定功能。一般由开发人员自行开发。客户软件不能直接访问 USB 设备,其与 USB 设备功能单元的通信必须经过 USB 系统软件和 USB 总线接口模块才能实现。客户软件一般包括 USB 设备驱动程序和界面应用程序两部分。USB 设备驱动程序负责和 USB 系统软件进行通信。通常,它向 USB 总线驱动程序发出 IRP(I/O Request Package)以启动一次 USB 数据传输。此外,根据数据传输的方向,它还应提供一个数据缓冲区以存储这些数据。

界面应用程序负责和 USB 设备驱动程序进行通信,以控制 USB 设备。它是最上层的软件,只能看到向 USB 设备发送的原始数据和从 USB 设备接收的最终数据。

(2) USB 系统软件。USB 系统软件负责和 USB 逻辑设备进行配置通信,并管理客户软件启动的数据传输。USB 逻辑设备是程序员与 USB 设备打交道的部分。USB 系统软件一般包括 USB 总线驱动程序和 USB 主控制器驱动程序这两部分。这些软件通常由操作系统提供,一般开发人员不必掌握。

(3) USB 总线接口。USB 总线接口包括主控制器和根集线器两部分。根集线器为 USB 系统提供连接起点,用于给 USB 系统提供一个或多个连接点(端口)。主控制器负责完成主机和 USB 设备之间数据的实际传输,包括对传输的数据进行串行编解码、差错控制等。该部分与 USB 系统软件的接口依赖于主控制器的硬件实现,一般开发人员不必掌握。

2. USB 设备(从机)

USB 协议中将 USB 设备定义为具有某种功能的逻辑或物理实体。在最底层,设备指一个独立的硬件部件;在较高层,设备表现为具有一定功能的硬件部件的集合,如一个 USB 接口设备;在更高层次上,设备是指连接到 USB 总线上的实体所具有的功能,如一个数据/传真调制解调器。总之,设备的含义可以是物理的、电气的、可寻址的或逻辑的。

USB 设备按照功能可分为两类:USB 集线器(Hub)和 USB 功能设备(USB 设备)。其中,USB 集线器主要用于为 USB 系统提供额外的连接点,它使得一个 USB 端口可以扩展连接多个 USB 设备;USB 设备可以通过 USB 集线器的下行端口与 USB 主机连接。如图 12-5 所示是 USB 集线器的逻辑图和实物图。

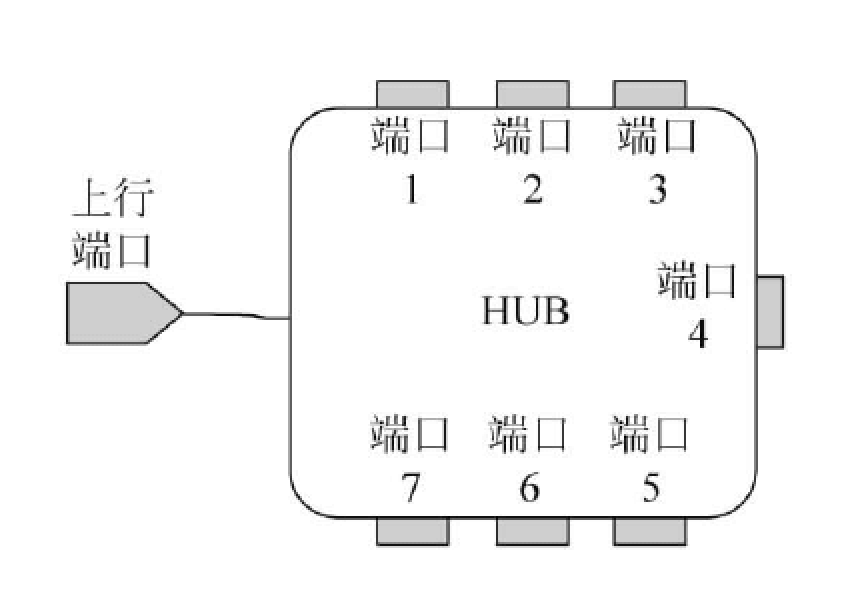




图 12-5　USB 集线器的逻辑图和实物图

USB 对一些具有相似特点并提供相似功能的设备进行抽象,进而将 USB 设备分成多种标准设备类,包括大容量存储设备类(MSD)、人机接口设备类(HID)、音频设备类(AUDIO)、视频设备类(VIDEO)、通信设备类(CDC)和集线器类(HUB)等。设备驱动程序通常由操作系统提供,开发人员可以直接使用,不必自己编写。设备描述符和接口描述符中

的类代码(CLASS)、子类代码(SUBCLASS)以及协议代码(PROTOCOL)指定了 USB 设备或其接口所属的设备类。表 12-1 给出了部分标准设备类的定义方法,如果要获得全部的标准设备类,请查阅 USB 实施者论坛上的类型规范文件。

表 12-1　标准的 USB 设备类举例

编　号	类 名 称	设备描述符 bDeviceClass 字段的值	设备描述符 bDeviceSubClass 字段的值	接口描述符 bInterfaceClass 字段的值	接口描述符 bInterfaceSubClass 字段的值
1	MSD	0x00	0x00	0x08	任意
2	HID	0x00	0x00	0x03	任意
3	AUDIO	0x00	0x00	0x01	任意
4	VIDEO	0xEF	0x02	0x0E	任意
5	CDC	0x02	0x00	0x02	任意
6	HUB	0x09	任意	0x09	任意
7	厂商定义类	0xFF	0xFF	0xFF	0xFF

12.1.3　USB 中断概述

为了理解如何使用 USB 驱动构件,需对 USB 中断有一定的了解,USB 模块的中断有 8 种类型,主要用到了复位中断、令牌完成中断和设备接入中断,见表 12-2。USB 模块中断详见 12.5.2 节。

表 12-2　USB 主要中断

中　断	功 能 描 述
ATTACH 中断	USB 主机检测到 USB 设备连接时触发,用于主机对设备的初始化
TOKDNE 中断	令牌处理完成时触发,用于 USB 设备设置或收发数据
USBRST 中断	USB 设备检测到有效的复位信号时触发

12.2　USB 设备(从机)的应用编程方法

USB 模块连接比较简单,只要将 USB 模块的两个引脚 USB_DP 和 USB_DM 分别接 33Ω 的电阻连接到 USB 接口的 D+和 D-即可。

12.2.1　USB 设备(从机)驱动构件及使用方法

1. USB 设备(从机)驱动要素分析

在将 KL25 作为 USB 设备时首先要初始化 USB 模块,为了完成与 PC 之间的枚举和数据传输,应具有初始化、枚举处理、发送数据和接收数据的基本功能。按照构件化设计思想,需有 4 个函数分别完成对应功能,分别是:初始化函数 usb_init 用以完成对 USB 模块的初始化配置、枚举函数 usb_enumerate 完成 USB 设备的枚举、数据发送函数 usb_send 和数据接收函数 usb_recv 完成数据的发送和接收。

2. USB 设备(从机)驱动构件头文件

```
//===========================================================
//函数名:usb_init
//功能: USB 模块初始
//参数:str:设备序列号
//返回: 无
//===========================================================
void usb_init(uint_8 str[]);

//===========================================================
//函数名: usb_enumerate
//功能: USB 枚举,用于处理 USB 设备复位后 USB 主机发送来的设备请求
//参数: 无
//返回: 无
//===========================================================
void usb_enumerate();

//===========================================================
//函数名: usb_send
//功能: USB 发送数据
//参数: buf: 待发数据缓冲区,并在函数返回时保留未发送的数据
//      len: 待发数据长度,并在函数返回时带出剩余未发送的数据长度
//返回: 实际发送的数据长度
//备注: 一次性传输的数据长度是端点所支持的最大数据长度(32B),如果发送的数据长度
//      大于 32B,则分为多次传输
//===========================================================
uint_8 usb_send(uint_8 * buf,uint_8 * len);

//===========================================================
//函数名: usb_recv
//功能: USB 接收数据
//参数: buf:接收数据缓冲区
//      len:函数返回时,带出接收的数据长度
//返回: 成功: 返回接收数据的长度; 失败: 返回 0
//===========================================================
uint_8 usb_recv(uint_8 * buf,uint_8 * len);
```

头文件中主要包含 USB 设备驱动要素分析之后的 4 个函数,以下是对这 4 个函数功能的详细介绍。

1) 初始化函数 void usb_init(uint_8 str[])

在使用 USB 设备之前,需要对 USB 模块进行初始化,主要包括分配 USB 模块使用的内存、USB 设备相关的描述符初始化、使能 USB 时钟源,以及使能 USB 中断等。初始化函数传入的参数是用户自定义的设备序列号,该设备序列号作为一个字符串描述符在 USB 设备枚举时传给 PC。

2) 枚举函数 void usb_enumerate()

该函数在 USB 设备枚举过程中执行,当 USB 设备发送来获取 USB 设备描述符请求、

配置USB设备请求时,由该函数进行处理和响应。

3) 发送数据函数 uint_8 usb_send(uint_8 * buf, uint_8 * len)

USB设备有数据要发送给USB主机,则可以使用该函数,并将发送的数据缓冲区的首地址和发送的数据长度作为参数传入即可,函数的返回值是发送成功的数据字节长度。

4) 接收数据函数 uint_8 usb_recv(uint_8 * buf, uint_8 * len)

接收数据函数用于接收USB主机发送来的数据,该函数传入的第一个参数buf是用于保存接收的数据的缓冲区地址,len是接收的数据长度。当USB主机发来数据时,该函数被调用,以实现数据的接收。该函数的返回值是接收成功的数据字节长度。

3. USB设备(从机)驱动构件使用方法

1) 在"includes.h"文件中声明全局变量位置声明USB接收和发送数组

```
//USB相关全局变量声明
uint_8   g_USBRecv[100];          //USB设备接收数据缓冲区,缓冲区最大32个字节数据
uint_8   g_USBRecvLength;         //USB设备接收数据长度
uint_8   g_USBSend[100];          //USB设备发送缓冲区,缓冲区最大32个字节的数据
uint_8   g_USBSendLength;         //USB发送数据长度
...
```

2) 在"main.c"文件中初始化USB设备

```
...
usb_init(Serial_String);      //USB设备初始化
...
```

3) 在需要的位置对接收或发送数组及其长度进行具体操作

```
...
g_USBSend[i]=g_USBRecv[i];
g_USBSendLength=g_USBRecvLength;
...
```

4) 在"isr.c"文件USB设备中断服务例程中调用接收或发送函数进行数据收发

```
...
bufferLen=usb_send(g_USBSend,&g_USBSendLength);
... ...
usb_recv(g_USBRecv,&g_USBRecvLength);
...
```

12.2.2 USB设备(从机)方MCU编程实例

1. USB中断程序文件

```
//==========================================
//文件名称:isr.c
```

```
//功能概要：中断底层驱动构件源文件
//版权所有：苏州大学恩智浦嵌入式中心(sumcu.suda.edu.cn)
//更新记录：2012-11-12    V1.0
//==========================================
#include "includes.h"

extern void usb_isr_handler(uint_8 isr_type);
extern uint_8  usb_get_isr();
extern uint_8 vEP2State;
extern tBDT * BDTtable;

//================中断函数服务例程================

//==========================================
//函数名称：USB0_IRQHandler(USB 中断服务例程)
//函数功能：处理 USB 模块中断,包括数据收发、设备枚举等原因触发的中断请求。
//          需收发数据时,会通过对全局变量的操作完成,如需发送数据,需预先将
//          待发送数据写入全局变量 g_USBSend,其长度写入 g_USBSendLength,并置
//          发送标志 g_USBSendFlag 为 1,中断会自动将数据发出；同时中断会自动接收
//          主机发来的数据,将数据放入全局变量 g_USBRecv,并将接收数据长度写入
//          全局变量 g_USBRecvLength
//==========================================

void USB0_IRQHandler(void)
{
    uint_8 isr_type;
    DISABLE_INTERRUPTS;                          //关总中断
    //1. 获取中断类型
    isr_type=usb_get_isr();
    //2. 若不是令牌完成中断,调用相应处理程序
    if(isr_type!=USB_TOKDNE_INT)
    {
        usb_isr_handler(isr_type);               //调用非令牌完成中断的处理程序
        goto USB0_IRQ_exit;
    }
    //3. 是令牌完成中断,执行块数据传输或设备枚举
    //(3.1)清收发 ODD 区,并指定 EVEN 区
    FLAG_SET(USB_CTL_ODDRST_SHIFT,USB0_CTL);
    //(3.2)若是设备枚举请求,进行设备枚举
    if((USB0_STAT >> 4)==0)
    {
        usb_enumerate();
        goto USB0_IRQ_exit;
    }
    //(3.3)若不是设备枚举请求,进行数据收发操作
    if((USB0_STAT & 0xF8)==mEP2_IN)              //从端点 2 发送数据至主机
    {
        //若执行发送标志被置 1,则执行发送数据函数
        if(g_USBSendFlag==1)
        {
```

```
                usb_send(g_USBSend, &g_USBSendLength);          //调用发送数据处理函数
                if(g_USBSendLength==0)
                {
                    g_USBSendFlag=0;                            //发送完成,执行发送标志清0
                }
                goto USB0_IRQ_exit;
            }
            //若无数据需要发送,则向USB主机发送一个确认包
            BDTtable[bEP2IN_ODD].Cnt =0;
            vEP2State ^= 0x40;
            BDTtable[bEP2IN_ODD].Stat._byte= vEP2State;
        }
        else if((USB0_STAT & 0xF8)==mEP3_OUT)               //从端点3接收主机发来的数据
        {
            usb_recv(g_USBRecv, &g_USBRecvLength);
        }
    USB0_IRQ_exit:
        //4. 清除中断标志,结束中断处理
        FLAG_SET(USB_ISTAT_TOKDNE_SHIFT, USB0_ISTAT);
        ENABLE_INTERRUPTS;                                  //开总中断
        return;
    }

    //==========================================================================
```

2. USB设备主程序文件

```
//说明见工程文件夹下的Doc文件夹内Readme.txt文件
//==========================================================================

#include "includes.h"                                //包含总头文件

//USB序列号
//注意:修改USB序列号时,要将序列号的第一个字节数据长度字段进行修改
uint_8 Serial_String[]=
{
    0x10,                                            //序列号长度字段(包含该字段本身长度)
    0x03,                                            //序列号索引
    //以下是用户自定义序列号
    'H',0x00,
    'W',0x00,
    'W',0x00,
    '_',0x00,
    'U',0x00,
    'S',0x00,
    'B',0x00,
};

int main(void)
```

```
{
    uint_32  mRuncount;                          //主循环计数器
    uint_8 i;
    //2. 关总中断
    DISABLE_INTERRUPTS;
    //3. 初始化外设模块
    light_init(RUN_LIGHT_BLUE, LIGHT_ON);        //蓝灯初始化
    usb_init(Serial_String);                     //USB 设备初始化
    //4. 给有关变量赋初值
    mRuncount=0;                                 //主循环计数器
    g_USBRecvLength=0;                           //USB 接收数据长度初始化
    g_USBSendLength=0;                           //USB 发送数据长度初始化
    g_USBSendFlag=0;                             //USB 执行发送标志初始化
    //5. 使能模块中断
    //6. 开总中断
    ENABLE_INTERRUPTS;
    //进入主循环
    //主循环开始==================================
    for(;;)
    {
        //运行指示灯(RUN_LIGHT)闪烁--------------------------------------
        mRuncount++;                             //主循环次数计数器+1
        if (mRuncount >= RUN_COUNTER_MAX)        //主循环次数计数器大于设定的宏常数
        {
            mRuncount=0;                         //主循环次数计数器清零
            light_change(RUN_LIGHT_BLUE);        //蓝色运行指示灯状态变化
        }
        //以下加入用户程序--------------------------------------------
        //USB 主机向 USB 设备发送要数据命令
        if(g_USBRecv[0]==cmdINTESTDATA && g_USBSendLength!=0)
        {
            g_USBRecv[0]=cmdNULL;
            g_USBSend[0]=cmdINTESTDATA;
            g_USBSendFlag=1;
        }
        //USB 主机发送数据,写入到发送数组中,以便主机要数据时进行发送
        else if(g_USBRecv[0]==cmdOUTTESTDATA && g_USBRecvLength!=0)
        {
            //禁止 USB 中断
            disable_irq(USB_INTERRUPT_IRQ);
            for(i=0;i<g_USBRecvLength;i++)
                g_USBSend[i]=g_USBRecv[i]+1;
            g_USBSendLength=g_USBRecvLength;
            g_USBRecv[0]=cmdNULL;
            g_USBRecvLength=0;
            //使能 USB 中断
            enable_irq(USB_INTERRUPT_IRQ);
        }
    }//主循环 end_for
    //主循环结束==================================
}
```

12.2.3　USB设备(从机)PC驱动问题

1. 驱动概述

设备驱动程序又叫功能驱动程序(Function Driver),它安装在计算机主机上,负责管理一种具体的设备。在Windows操作系统下,应用程序不能直接对硬件接口操作,对硬件资源的访问是交由驱动程序完成的。应用程序只要同设备驱动程序通信,就能实现和设备的数据交换。

设备驱动程序只是支持计算机主机与设备进行通信的软件组中的一层。因为在USB通信中使用分层式的驱动程序模型,其中最有代表性的是Microsoft力推的全新驱动程序模式WDM(Windows Driver Model),即Windows驱动程序模型(关于WDM,请参阅其他相关驱动开发参考文献,这里不再详述)。不同层之间的驱动程序完成不同的操作,且可相互调用。驱动程序分为以下几个层次：总线驱动程序、总线过滤驱动程序、下层过滤驱动程序、功能驱动程序和上层过滤驱动程序。总线驱动程序负责管理逻辑的或物理的总线,例如PCMCIA、PCI、USB、IEEE 1394和ISA,检测并向PnP(Plug and Play,即插即用)管理器通知总线上的设备,并且能够管理电源。过滤驱动程序(Filter Driver)与功能驱动程序协同工作,用于增加或改变功能驱动程序的行为。一般操作系统已经提供了总线驱动和过滤驱动。

有时候用户USB设备的功能在Windows提供的驱动程序或者厂商提供的驱动程序中都没有,这时候就需要自己编写驱动程序。

2. 驱动程序开发工具

要建立一个WDM驱动程序通常需要Microsoft的Visual C++编译器(其中包含完整的开发环境和调试功能)、Windows DDK(Windows Device Developer's Kit,Windows设备开发包)、MSDN(Microsoft's Developer's Network)即Windows面向软件开发者的一种信息服务、驱动程序工具软件以及高级的调试器等。

其中,Windows DDK包含以下内容。

(1) 范例代码和文件说明。与USB相关的文件包括WDM驱动程序的教学示范,以及USB驱动程序源代码。

(2) 批量传输的源代码和编译码、文件说明以及bulkusb.sys驱动程序的范例应用程序。bulkusb.sys驱动程序可以使用在任何支持批量传输的USB芯片内,应用程序使用ReadFile和WriteFile函数来读/写数据。

(3) 处理实时传输的isousb.sys驱动程序。在USB Implementers Forum网站上,有使用这些驱动程序的技巧和修正说明。

(4) 一个过滤器驱动程序范例,以及USBView工具软件。

3. 驱动标识GUID

GUID(Globally Unique Identifier,全球唯一标识符)是一个128位的字母数字标识符,用于指示产品的唯一性安装。在许多流行软件应用程序(例如Web浏览器和媒体播放器)中,都使用GUID。

GUID的标准格式为"xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx",将GUID分成5个类别的十六进制字符,类别之间以连字号"-"隔开,其中,x是0～9或A～F范围内的一个十六进制的数字。例如,6F9619FF-8B86-D011-B42D-00C04FC964FF就是一个有效的

GUID 值。

世界上任何两台计算机都不会生成重复的 GUID 值。GUID 主要用于在拥有多个结点、多台计算机的网络或系统中，分配具有唯一性的标识符。在 Windows 平台上，GUID 应用非常广泛，如注册表、类及接口标识、数据库，甚至自动生成的机器名、目录名等。

这里用 GUID 唯一标识驱动程序，是主机应用程序和驱动程序的通信接口（如主机应用程序可以通过驱动的 GUID 找到设备路径）。Windows 会为标准对象（例如 HID 类别）定义 GUID。其他设备可以使用 Visual Studio 编译环境内所附的 guidgen.exe 程序，来取得一个 GUID。如在 VC.NET 2003 环境下，选择菜单栏中“工具”下的“创建 GUID”选项，就可以弹出如图 12-16 所示的“创建 GUID”对话框，以生成新的 GUID。GUID 包含在驱动程序的程序代码内，应用程序可以使用 API 来读取它。

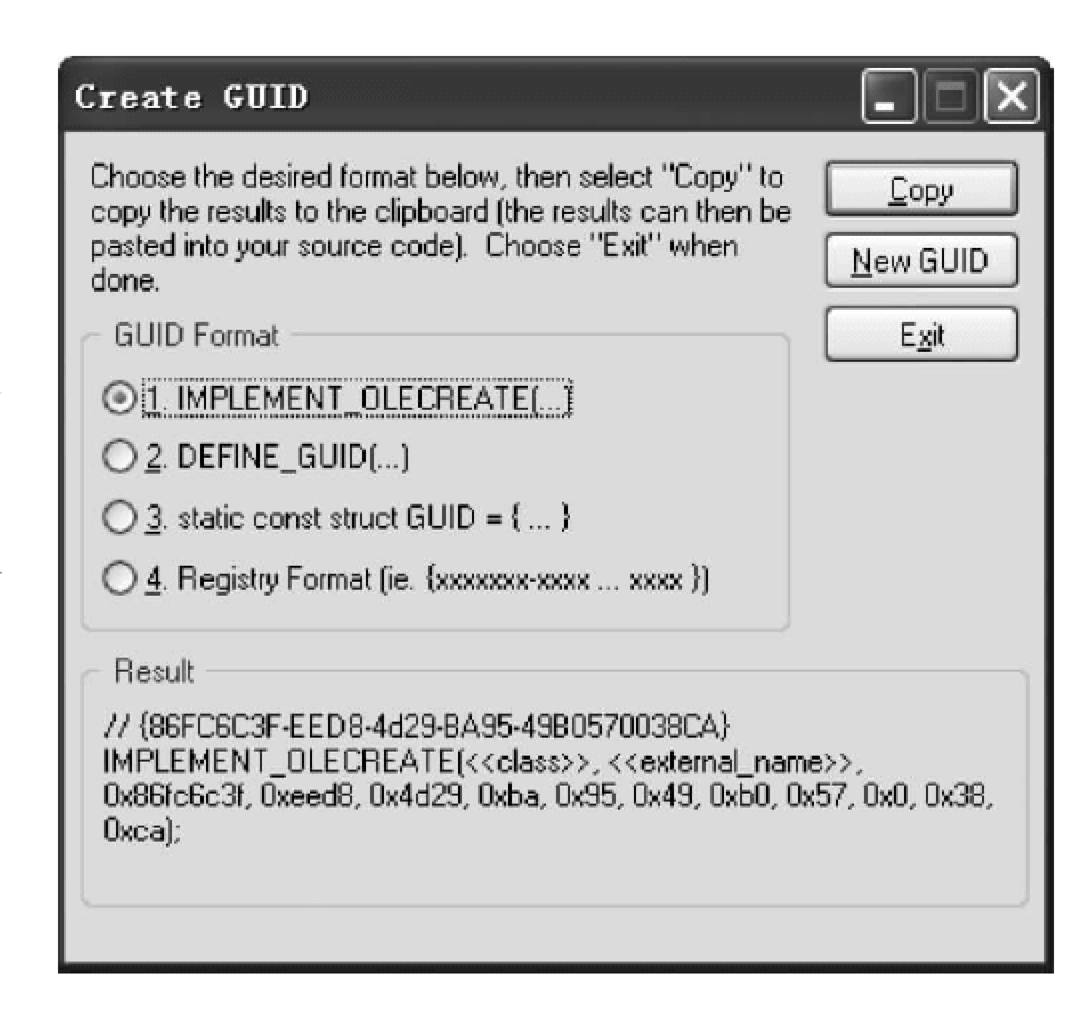


图 12-6　“创建 GUID”对话框

4. 驱动程序文件(.sys)和设备信息文件(.inf)

驱动程序文件是驱动程序的可执行代码，其扩展名为.sys。对于 PnP 设备，在设备插入后，.sys 文件会被 Windows 装载到内存中，系统线程调用.sys 中的函数来和设备进行通信。驱动程序编写好了之后，用 DDK 工具可以生成相应的.sys 文件。

INF(Device Information File，设备信息文件)用来指示安装设备驱动程序(*.sys)，驱动程序要配合 INF 才可以安装。INF 文件的后缀名为.inf，但实际上是一个文本文件，可以用记事本工具打开、查看和编辑。在 INF 文件中提供了设备的产品 ID、对应的.sys 文件名、驱动 class 名以及 class GUID；还指明了硬件驱动该如何安装到系统中、源文件在哪里、安装到哪一个文件夹中、怎样在注册表中加入自身相关信息等。INF 文件应该由驱动程序开发人员随驱动程序一起提供。用户可以按照一定的语法规则自行编写 INF 文件，但为了快速开发，也可在 Microsoft 提供的模板上修改，得到合适的 INF 文件。

INF 文件由许多按层次结构排列的节组成。INF 文件中常用的节如表 12-3 所示。

表 12-3　INF 文件中常用的节

节	说　明
Version	INF 文件的开始，描述了版本信息，主要用于版本控制
DestinationDirs	指明 INF 文件和驱动程序的目标目录
SourceDisksNames	指明驱动程序所在的磁盘或 CD-ROM
SourceDisksFiles	指明驱动程序的文件名
Manufacturer	指明供应商及其对应 Models 节的名称
DDInstall(Windows 2000)	指明需复制的文件、向注册表中添加的内容等信息
DDInstall.Services	指明驱动程序安装的详细信息

在Version节,根据驱动使用的具体GUID,对ClassGUID进行赋值。在Manufacturer节,指定了设备的VID和PID,在设备固件的设备描述符中也定义了这两个值。当设备固件中的这两个值改变后,这里也要做相应修改。这样才能保证PC连接设备后,能正确加载合适的驱动。在Strings段定义了设备驱动的描述信息。

5. 驱动程序的安装

在设备的枚举过程中,当主机获得完整的配置描述符之后,并在主机对设备配置之前,主机弹出Windows消息,显示"产品字符串",如图12-7(a)所示,产品字符串可以在固件程序的任意一个字符串描述符中定义,再将设备描述符的产品字符串索引域定义为相应的字符串索引即可。然后,主机开始分配并加载设备驱动。主机通过设备描述符获悉了设备后,会为该设备寻找一个最合适的驱动来管理与设备的通信。在选择一个驱动的时候,Windows尝试将设备中的VID、PID和版本号(此项可选)与PC的INF文件中的相匹配,逐个寻找与之相符合的INF文件。过程如下:①系统从连接的USB设备中获得设备的VID(设备描述符的供应商字段)和PID(设备描述符的产品字段),从而得到设备的硬件ID。②系统查找与该硬件ID相符的INF文件,如果找不到,系统将读取接口描述符,从中提取USB设备的兼容ID,并查找与兼容ID相符合的INF文件。③如果仍未找到,系统提示用户自己安装该USB设备驱动程序。这时,会显示"找到新的硬件向导"对话框,引导用户安装相应的设备驱动程序。一般只要从指定位置安装.sys后缀的驱动程序即可。

在操作系统成功安装驱动程序之后,系统会将驱动程序文件*.sys保存在windows/system32/drivers目录中,将相应的*.INF文件保存在windows/inf目录中(Windows 2000/XP会将没有经过微软认证的驱动程序的INF文件改名为OEMx.INF和OEMx.PNF来存放,x是从数字0开始的正整数,PNF为预编译信息文件),并且系统会弹出消息,提示用户"新硬件已安装并可以使用了",如图12-7(b)所示。同时Windows"设备管理器"中会增加这个已安装的新设备驱动项,如图12-7(c)中的SoochowUniversity-USBDevice所示。驱动程序安装完成后,系统的注册表也会储存所安装设备驱动的信息。Windows通过注册表和INF文件来将设备和驱动程序之间的关系绑定。单击Windows桌面上的"开始"菜单,选择"运行",在"运行"对话框中输入"regedit",确定后就可以查看和编辑注册表的内容了。如图12-7(d)所示就是一个Windows注册表。注册表编辑器使用树状结构来安排其内容,选择HKEY_LOCAL_MACHINE|SYSTEM|ControlSet001|Control|DeviceClasses分支,在该分支下再选中设备驱动程序的GUID,就可以显示USB设备的驱动程序的注册信息。

如果主机已经安装了设备使用的驱动程序,这时再接入相应的设备,会有一连串的配置过程,这个过程是由系统自带的驱动程序来完成的,对用户来说是透明的。在配置的过程当中,系统会根据USB设备返回来的信息(PID和VID)到注册表中寻找相应的驱动程序。具体步骤如下:①USB总线报告设备改变;②系统发送I/O请求包(IRP),获得子DeviceObject,得到硬件ID、兼容ID等;③为子Device设定PhysicalDevice信息;④检查是否已经安装过驱动程序。如果已经安装过,就直接加载驱动;否则,检查驱动程序数据库,查看有没有与硬件ID、兼容ID相同的驱动程序,如果有,则直接安装;否则报告"找到新硬件"。

本书配套网上教学资源中已配备本USB设备Windows驱动,用户无须自行设计,直接



图 12-7 Windows 中的设备信息

安装使用即可。

12.2.4 与 USB 设备(从机)通信的 PC 方程序设计

通常,PC 方应用程序作为主机程序要有数据接收、处理和发送能力,并能够提供友好的界面以便于用户操作和使用。

1. MCU 方测试工程的功能

测试工程功能概述如下。

(1) 将 PC 与 KL25 的 USB 接口相连接,等待 PC 方设备驱动程序安装完成,安装过程可参考 12.2.4 节。

(2) 主循环中,改变蓝灯的状态(蓝灯一直闪烁)。

(3) 打开 PC 方测试程序,可以看到枚举到的设备的信息。PC 通过 USB 向 MCU 发送数据,MCU 方将接收到的数据保存到全局变量。

(4) PC 方单击"接收数据"按钮,向 KL25 发送要数据的命令,KL25 再将保存到全局变量中的数据读出来并发给 PC。

USB 构件的测试工程位于网上教学资源中的"..\program\CH12_KL25-USB"文件夹。

2. PC 方测试工程的功能

为了让读者掌握 USB 通信机理,本书配套的光盘中提供了一个 PC 方测试程序。测试程序采用 C# 语言编写,工程主要包括一个窗体模块(USBTest.cs)和一个标准模块(USB.cs)。

测试程序的运行界面如图 12-8 所示。该测试程序不仅提供了基本的 USB 数据发送和接收功能,还能实时显示程序运行时的状态、错误信息以及目标设备的相关描述符。用户可在发送框中输入数据,再单击"发送"按钮,将数据通过 USB 总线发送给低端目标设备,此时程序使能"接收"按钮,单击该按钮后,PC 方程序将从 USB 设备收回上一次发送的数据,并显示在接收数据文本框中。



图 12-8　PC 方测试程序界面

12.3　USB 主机的应用编程方法

12.3.1　USB 主机驱动构件及使用方法

1. USB 主机驱动要素分析

根据需要完成的功能,USB 主机驱动构件主要应包括 USB 主机初始化、USB 接入设备初始化、从 USB 设备读取数据、向 USB 设备写入数据以及检测 USB 设备的连接状态。

按照构件化设计思想将这些功能封装成 5 个函数,USB 主机初始化和接入设备初始化函数由 USBHostInit 函数和 InitUSBDevice 函数完成,发送和接收数据函数由 USBReadData 函数和 USBWriteData 函数完成,检测 USB 设备状态由 CheckUSBDeviceStatus

函数完成。

2. USB 主机驱动构件头文件

```
//==========================================
//函数名:USBHostInit
//功能: USB 模块初始
//参数: 无
//返回: 0=成功; 非 0=异常
//==========================================
uint_8 USBHostInit(void);

//==========================================
//函数名:InitUSBDevice
//功能: 初始化接入的 USB 设备
//参数: device_inf:函数返回时带回初始化的 USB 设备信息
//返回: 0=成功; 1=异常
//==========================================
uint_8 InitUSBDevice(uint_8 * device_inf);
//==========================================
//函数名: USBReadData
//功能:  USB 数据读取
//参数:  ep:USB 端点号
//         length:读的数据长度
//         ReadBuffer:存放读数据的缓冲区
//返回: 0=成功; 1=失败
////备注: 调用内部函数 USBStartTransaction 执行数据的读取,读取的数据可以在多个事
//        务处理中,因此需要根据端点所支持的最大数据包的长度决定执行多少次事务处理
//==========================================
uint_16 USBReadData(uint_8 ep, uint_16 length,uint_8 *  ReadBuffer);

//==========================================
//函数名: USBWriteData
//功能:  向 USB 设备(U 盘)写入数据
//参数: ep:USB 端点号
//        length:要写入的数据长度
//        buff:存放要写入的数据的缓冲区
//返回: 0=成功; 1=失败
//备注: 调用内部函数 USBStartTransaction 执行数据的写入,写入的数据可以在多个事
//        务处理中,因此需要根据端点所支持的最大数据包的长度决定执行多少次事务处理
//==========================================
uint_8 USBWriteData(uint_8 ep,uint_16 length,uint_8 * WriteBuffer);

//==========================================
 //函数名: CheckUSBDeviceStatus
 //功能:  检测 USB 设备状态
 //参数: 无
 //返回: 0=成功; 1=失败
//==========================================
uint_8 CheckUSBDeviceStatus();
```

下面是这5个函数的详细功能介绍。

1) USB模块初始化函数 USBHostInit(void)

在使用USB主机之前,需要对USB模块进行初始化,主要包括分配USB模块使用的内存、USB相关端点初始化、使能USB模块时钟源、使能USB中断、设置BDT寄存器以及使能USB主机模式等。初始化函数执行完之后,KL25就一直等待U盘插入到USB端口,若此时有U盘插入,则KL25就会检测到U盘,并开始对U盘进行枚举,获取U盘的相关信息。在枚举完成之后,KL25就可以与U盘进行数据的传输。

2) USB接入设备初始化函数 InitUSBDevice(uint_8 * device_inf)

该函数是在USB主机初始化完成之后执行,该函数一直等待是否有设备连接。如果有USB设备连接,则开始对USB设备进行枚举。

3) 数据读取函数 USBReadData(uint_8 ep, uint_16 length,uint_8 * ReadBuffer)

该函数用于KL25向U盘读数据,调用文件系统层函数 znFAT_ReadData,通过传入的参数计算出要读取的扇区数目,然后调用USB类层函数 USB_Class_Read_Sector 对要读取的扇区进行处理,接着调用USB设备层函数 usb_read_sector,该函数开始组合要发送给USB设备的命令,然后在 usb_device_transfer 中调用USB驱动构件层函数 USBReadData 进行USB事务处理,将进行命令的发送和数据的读取。

4) 数据写入函数 USBWriteData(uint_8 ep,uint_16 length,uint_8 * WriteBuffer)

数据写入的执行过程和数据读取的执行过程类似,最后调用USB驱动构件层的函数 USBWriteData 执行数据写入的事务处理。

5) 检测USB设备连接状态函数 CheckUSBDeviceStatus

在USB设备连接到KL25之后,KL25需要获取USB设备的状态,以判断USB设备是否处于连接中,从而执行相应的处理。

3. USB主机驱动构件的使用方法

1) 定义全局变量以保存USB从机连接标志

```
…
uint_8 USBFlag = 0;
…
```

2) 在"main.c"文件中初始化USB主机

```
…
USBHostInit();              //USB主机初始化
…
```

3) 在文件isr.c的USB中断处理函数中改变USBFlag变量以标示USB从机状态

```
…
USBFlag |= USB_ISTAT_ATTACH_MASK;
…
```

4）在需要的位置检测到 USBFlag 变化，然后对接入 USB 设备进行初始化

```
...
if(USBFlag & USB_ISTAT_ATTACH_MASK)
{
...
    if(InitUSBDevice(Device_INF) == 0)
    {
    ...
    }
}
...
```

5）通过文件系统函数间接调用 USBReadData 或 USBWriteData 与 USB 从机通信

```
...
znFAT_ReadData(&fileinfo1,0,13,buf);
...
znFAT_Close_File(&fileinfo1);
...
znFAT_Flush_FS();              //刷新文件系统
...
```

12.3.2　USB 主机方 MCU 编程实例

1. USB 主机主程序文件

```
//说明见工程文件夹下的 Doc 文件夹内 Readme.txt 文件
//==========================================================

#include "includes.h"                                   //包含总头文件

//USB 相关全局变量定义
uint_8 USBHostStatus = USB_DEVICE_IDLE;                 //USB 设备状态
uint_8 USBFlag = 0;
uint_8 UartBuff[128] = {0};
uint_8 UartBuffLength = 0;
uint_8 UartBuffFull = 0;
uint_8 DeviceInserted = 0;
struct znFAT_Init_Args Init_Args;
struct FileInfo fileinfo1,fileinfo2;
const char Text[]="KL25 USB Mass Storage Host Controller. Date: 2016.04.16";
uint_8 Device_INF[32];

int main(void)
{
    //1. 声明主函数使用的变量
```

```
uint_32  mRuncount;                                  //主循环计数器
uint_32  Count = 1;
uint_8 buf[14];
uint_8 i;
uint_8 err;
//2. 关总中断
DISABLE_INTERRUPTS;

//3. 初始化外设模块
light_init(RUN_LIGHT_BLUE, LIGHT_ON);                //蓝灯初始化
uart_init(UART_1, 9600);                             //使能串口 1,波特率为 9600
USBHostInit();                                       //USB 主机初始化
printf("------------------------------USB 主机测试!----------------------\r\n");
//4. 给有关变量赋初值
mRuncount=0;                                         //主循环计数器
for(i=0; i<13; i++)
    buf[13]=0;
buf[12]='\0';
//5. 使能模块中断
uart_enable_re_int(UART_1);                          //使能串口 1 接收中断
//6. 开总中断
ENABLE_INTERRUPTS;
printf("System started, Please insert your USB MSD!\r\n");
//初始化接入 USB 设备
if(InitUSBDevice(Device_INF) == 0)
        USBHostStatus = USB_DEVICE_IDLE;             //没有设备连接
    else
    {
        USBHostStatus = USB_DEVICE_CONNETED;  //设备连接
        Count = 1;
        printf("Device connection detected.\r\n");
        printf("Mass-storage driver started.\r\n");
        printf("USB MSD information: %s\r\n",Device_INF);
    }

//进入主循环
//主循环开始==================================
for(;;)
{
    //运行指示灯(RUN_LIGHT)闪烁--------------------------------------------
    mRuncount++;                                 //主循环次数计数器+1
    if (mRuncount >= RUN_COUNTER_MAX)    //主循环次数计数器大于设定的宏常数
    {
        mRuncount=0;                             //主循环次数计数器清零
        light_change(RUN_LIGHT_BLUE);
                                     //蓝色运行指示灯(RUN_LIGHT_BLUE)状态变化
    }

    if(Count > 1000000)
```

```
        {
            Count = 0;
            if(USBHostStatus == USB_DEVICE_CONNETED)
            {
                znFAT_Device_Init();
                znFAT_Select_Device(0, &Init_Args);
                err=znFAT_Init();              //znFAT 初始化
                if(err)
                {
                    DeviceInserted = 0;
                    printf("\r\nFile system init failed!\r\n");
                    while(1);
                }
                printf("\r\nFile system init ok!\r\n");
                DeviceInserted = 1;            //连接的 USB 设备数目
            }
        }
        if(Count)
        {
            Count++;
        }

        if(DeviceInserted == 1)
        {
            err= znFAT_Open_File(&fileinfo1,"/suda.txt",0,1);
            znFAT_ReadData(&fileinfo1,0,13,buf);
            printf("Read File OK!\n");
            printf("Udisk Data:%s\n",buf);
            znFAT_Close_File(&fileinfo1);
            printf("Close File OK!\r\n");
            znFAT_Flush_FS();                  //刷新文件系统
            DeviceInserted=0;
        }
        if(USBFlag & USB_ISTAT_USBRST_MASK)
        {
            USBFlag &= ~USB_ISTAT_USBRST_MASK;
            if(CheckUSBDeviceStatus() == 1)
            {

            }
            else
            {
                if(USBHostStatus == USB_DEVICE_CONNETED)
                {
                    DeviceInserted = 0;
                    printf("Device removed.\r\n");
                    USBHostStatus = USB_DEVICE_DETACHED;
                    USBHostInit();
                }
            }
```

```
            }
            if(USBFlag & USB_ISTAT_ATTACH_MASK)
            {
                USBHostStatus &= ~USB_DEVICE_ATTACHED;
                if(InitUSBDevice(Device_INF) == 0)
                {
                    USBHostStatus = USB_DEVICE_IDLE;
                }
                else
                {
                    USBHostStatus = USB_DEVICE_CONNETED;
                    Count = 1;
                }
            }
            //以下加入用户程序-----------------------------------------------------
        }//主循环 end_for
        //主循环结束==================================
}
```

2. USB主机测试实例

测试工程功能概述如下。

(1) 串口通信格式:波特率9600,1位停止位,无校验。

(2) 蓝色灯一直保持亮的状态。

(3) 通过USB转接线将U盘插入,在串口调试工具上会看到枚举到的设备信息,不同型号的U盘枚举到的设备不同。另外,还可读取并显示U盘文件内的预存的数据“suda-usb-test”。

USB构件的测试工程位于网上教学资源中的“..\program\CH12_KL25-USB”文件夹。

12.4　设计微控制器的USB驱动构件应掌握的基础知识

12.4.1　USB底层编程涉及的基本概念

1. USB端点概念及特征

USB设备中唯一可以寻址的就是USB端点,主机和设备之间的通信最终作用于设备的端点上,端点是主机与设备之间通信的结束点。根据端点的用途不同,可将端点分为两类:0号端点和非0号端点。0号端点比较特殊,被称为控制端点,它既支持上行传输(IN)又支持下行传输(OUT),且只用在控制传输,其他端点只能在单方向传输数据。设备中的每个端点都有一个唯一的端点号,和端点的方向构成一个该端点的地址(8位)。

控制端点在USB设备上电复位以后就可以使用,而其他端点必须要在配置以后才可以使用。根据具体应用的需要,USB设备还可以含有多个除控制端点以外的其他端点。对于低速设备,其附加的端点数最多为4个,端点号范围是0～3;对于全速/高速设备,其附加的

端点数最多为 16 个,端点号范围是 0～15。端点的传输特性决定了其与主机通信时所采用的传输类型,如控制端点只能使用控制传输。端点具有以下特性。

(1) 端点的总线访问频率;

(2) 端点的总线延迟;

(3) 端点的带宽;

(4) 端点的端点号;

(5) 错误处理;

(6) 端点所支持的最大数据包的长度;

(7) 端点的传输类型;

(8) 端点的数据传输方向。

2. USB 通信管道

USB 数据是通过管道传输的,在传输发生之前,主机和设备之间必须先建立一个管道。USB 管道并不是一个实际对象,它只是设备端点和主机控制器软件之间的关联,代表了一种在两者之间移动数据的能力。每个 USB 设备都有一个默认的控制管道,用于控制传输。该管道是在设备连接到主机并成功复位后建立的,其他的管道是在设备被配置后才存在的。如果设备从总线上移除,主机也就撤销这个不再使用的管道。端点和各自的管道在每个方向上按照 0～15 编号,因此一个设备最多有 32 个活动管道,即 16 个 IN 管道和 16 个 OUT 管道。主机和设备之间的通信管道示意图如图 12-9 所示。

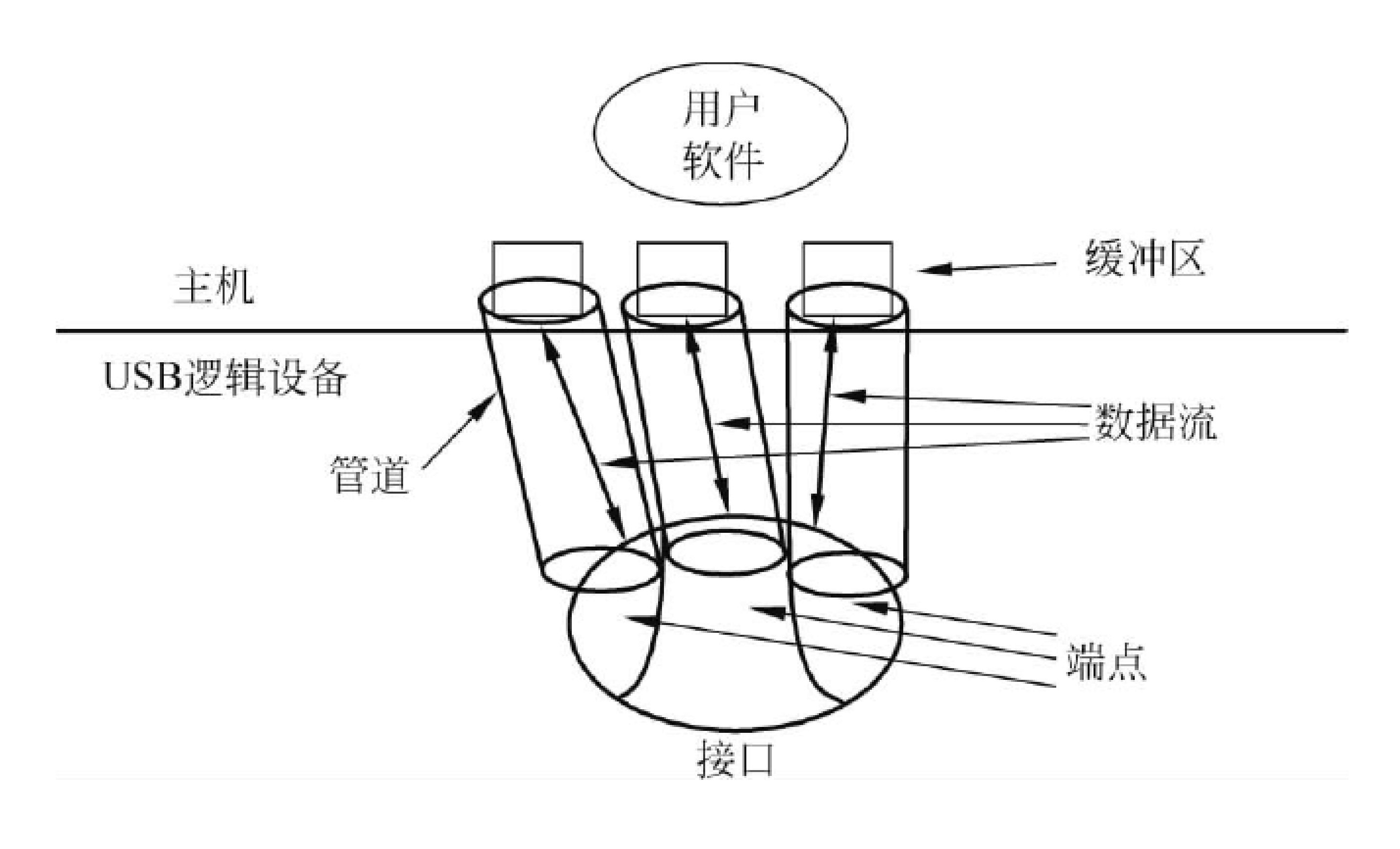


图 12-9 管道

3. USB 基本通信单元:包

在 USB 协议中,包(Package)是 USB 系统中数据传输的基本单元,所有数据都是经过打包后在总线上传输的。USB 数据传输中的包类型包括令牌包、数据包和握手包等。

1) 包字段格式

包由一些字段构成,不同功能的字段,按照特定的格式组合可以构成不同的包。包的字段主要有同步字段(SYNC)、包标识符字段(PID)、地址字段(ADDR)、端点字段(ENDP)、帧号字段、数据字段、校验字段(CRC)和包结尾字段(EOP)。每种包的格式不尽相同,下面对各个字段的含义进行介绍。

(1) 同步字段(SYNC)。同步字段由 8 位组成,作为每个包的前导。顾名思义,它是起

同步作用的，目的是使USB设备与总线的包传输速率同步，其值固定为00000001B。

(2) 包标识符字段(PID)。包标识符字段(PID)紧随在SYNC字段后面，用来表示包的类型。在USB协议中，根据PID的不同，包具有不同的类型、格式和含义。该字段各个位如表12-4所示。

表12-4　PID字段

D0	D1	D2	D3	D4	D5	D6	D7
PID0	PID1	PID2	PID3	$\overline{\text{PID0}}$	$\overline{\text{PID1}}$	$\overline{\text{PID2}}$	$\overline{\text{PID3}}$

PID字段长度为一个字节(8位)，由4个包类型位和4个校验位构成。PID是包类型的唯一标识，主机和设备在接收到包后，首先必须对包标识符解码得到包的类型。PID中的校验字段是通过对类型字段的每位取反得到的，它用于对包类型字段进行错误检查，旨在保证包中标识符的可靠性。如果4个校验位不是它们各自的类型位的反码，则说明标识符中信息有错误。

(3) 地址字段(ADDR)。设备地址由7位组成，共有128个地址值。地址0作为默认的地址，不能分配给USB设备，因此只有127个可配置的地址值。在设备上电时，主机用默认地址0与设备通信。当设备上电配置完成后，主机重新为设备分配一个地址。

(4) 端点字段(ENDP)。端点字段由4位组成，可寻址16个端点。该字段仅用在IN、OUT和SETUP令牌包中。低速设备可支持端点0以及端点1作为中断传输模式，而全速和高速设备则可以包含全部的16个端点。

(5) 帧号字段。帧号字段仅存在于SOF包中，长度为11位，从0开始取值，每发送一个SOF，该字段值自动加1，最大值为0x7FF(十进制2047)，当超过最大值时自动从0开始循环。

(6) 数据字段。数据字段的最大长度为1024字节，在数据传输时，首先传输低字节，然后传输高字节。对于每一个字节，先传输字节的低位，再传输字节的高位。实际的数据字段的长度，需根据设备的传输速率(低速、全速或高速)以及传输类型(中断传输、批量传输、同步传输和控制传输)而定。

(7) 校验字段(CRC)。校验字段采用的是循环冗余校验，一般在发送方的位填补操作之前进行，这样可以检验包是否存在错误，保证传输的可靠性。在令牌包中，一般采用5位循环冗余校验；在数据包中，采用16位循环冗余校验。

(8) 包结尾字段(EOP)。USB主机根据包结尾字段P判断包的结束。对于低速和全速设备，包结尾字段在物理上表现为两位的SE0状态和一位的J状态。对于高速设备，通过故意填充位错误来表明包结束字段，例如，除了SOF的包之外，包结束字段表现为没有位填充的01111111的NRZ序列，也就是说，如果在EOP字段之前最后一个位是J状态，那么EOP字段表现为8个K状态。

2) 包类型

包类型由PID字段表示，表12-5给出各种类型的令牌包、数据包和握手包对应的PID字段的值。

表 12-5　PID 字段

包　类　型	PID 名称	PID 值(D0:3)	说　　明
令牌包	SETUP	1101B	通知 USB 设备将要开始一个控制传输
	IN	1001B	通知设备将要输入数据
	OUT	0001B	通知设备将要输出数据
	SOF	0101B	通知设备这是一个帧起始包
数据包	DATA0	0011B	不同类型的数据包
	DATA1	1011B	
	DATA2	0111B	
	MDATA	1111B	
握手包	ACK	0010B	成功接收数据确认包
	NAK	1010B	数据未准备好
	STALL	1110B	端点挂起或不支持该类型传输
	NYET	0110B	数据接收方未准备好
特殊类	PRE	1100B	SPLIT 传输前导(令牌包)
	ERR	1100B	SPLIT 传输错误(握手包)
	SPLIT	1000B	分裂事务(令牌包)
	PING	0100B	PING 测试(令牌包)
	—	0000B	保留,未使用

(1) 令牌包

在 USB 协议中定义了 7 种令牌包,其中最常见的 4 种是 SETUP 令牌包、OUT 令牌包、IN 令牌包、SOF 令牌包。

SETUP 令牌包、OUT 令牌包和 IN 令牌包的格式为:

同步字段 (SYNC)	包标识符字段 (PID)	地址字段 (ADDR)	端点字段 (ENDP)	校验字段 (CRC5)	包结尾字段 (EOP)

SOF 令牌包的格式为:

同步字段(SYNC)	包标识符字段(PID)	帧号字段	校验字段(CRC5)	包结尾字段(EOP)

在 USB 协议中,只有主机才能发送令牌包。令牌包定义了数据传输的类型,令牌包中最重要的是 SETUP、IN 和 OUT 令牌包,用来建立一次数据传输。IN 令牌包用来建立设备到主机之间的数据传输,OUT 令牌包用来建立主机到设备的数据传输。IN 和 OUT 令牌包可以对任何设备上的任何端点寻址。SETUP 令牌包是一个特殊的 OUT 令牌包,总是指向设备的端点 0,它是“最高优先级的”,也就是说设备必须先接收它,即使设备正在进行数据传输操作也要对其进行响应。

在低速和全速模式下,主机每间隔 1ms(1ms 之间被称为一帧,允许误差 0.005ms)发送一个帧开始包(Start of Frame,SOF),可以把帧理解为这 1ms 时间段内传输的信息。在高速模式下,主机每隔 125μs(一个微帧开始包,允许误差 0.0625μs)发送一个帧开始包,可以看到一个微帧开始包的周期是一个帧开始包的 1/8。

(2) 数据包

常用的数据包有两种类型,根据包标识符字段的不同,数据包分为DATA0数据包和DATA1数据包,这样做是为了支持数据切换同步。主机在控制传输的设置阶段发送的数据包总是DATA0数据包,在数据阶段发送的首个数据包必须是DATA1数据包。在数据阶段若有后续的数据包,则数据阶段的下一个则是DATA0数据包,之后将在DATA1数据包和DATA0数据包之间切换。数据包格式为:

同步字段(SYNC)	包标识符字段(PID)	数据字段(0~1024B)	校验字段(CRC16)	包结尾字段(EOP)

(3) 握手包

握手包仅由三个字段构成,用来报告事务处理的完成状态,事务处理的概念在下一部分介绍。握手包的格式为:

同步字段(SYNC)	包标识符字段(PID)	包结尾字段(EOP)

ACK握手包:当数据传输的接收方正确接收到数据包的时候,接收方将返回ACK握手包。ACK握手包表示一次正确的事务处理,之后才可以进行下一次事务处理。对于IN事务处理,ACK握手包由主机发出;对于OUT事务处理,ACK握手包由设备发出。

NAK握手包:对于IN事务处理,NAK握手表示设备没有传输数据到主机;对于OUT事务处理,NAK握手包由设备发给主机,表示主机没有传输数据到设备。NAK握手包仅能由设备发出,主机决不能发出NAK握手包。另外,出于流控制的目的,NAK握手包表示设备暂时不能接收和发送数据,但是能够在不需要主机干涉的情况下恢复数据传输。

STALL握手包:STALL握手包表示设备不能接收或者发送数据。对于IN事务处理,主机给设备发送IN令牌包后,若设备无法给主机发送数据,则直接给主机发送一个STALL握手包;对于OUT事务处理,主机给设备发送OUT令牌包和数据包后,若设备无法接收主机发来的数据,则直接给主机发送一个STALL握手包。STALL握手包仅能由设备发出,在任何条件下都不允许主机发送STALL握手包,返回STALL握手包的设备的状态是没有定义的。

4. USB的事务处理

USB事务处理是USB主机和USB设备之间通信的基础,而事务是由包组成的。一次事务处理(Transaction)由一个令牌包、一个数据包和一个握手包组成。令牌包表示事务处理的开始,所有的事务处理都必须包含令牌包,令牌包的类型决定了事务处理的类型,USB协议中定义了7种令牌包,因此有7种类型的事务处理,最重要的三种事务处理是SETUP事务处理、IN事务处理和OUT事务处理,令牌包包含事务类型、传输方向、地址和端点号;数据包包含本次事务处理要传输的数据,依照令牌包中的传输方向,USB主机或设备发送数据包,当然一次事务处理可能不需要发送任何数据,此时就不会发送数据包;握手包用于数据的接收方向发送方报告此次事务处理是否成功。

1) SETUP事务处理

SETUP事务处理是一种特殊的事务处理,其只在控制传输中的设置阶段使用,用于对设备进行配置,并且SETUP事务处理的数据传输方向总是主机到设备。在实际传输过程

中，难免会出现各种错误，如果接收到的 SETUP 令牌包中有错误，设备将忽略该包，不做任何响应。一个完整正确的 SETUP 事务处理流程如图 12-10 所示。



图 12-10　完整的 SETUP 事务处理

2）IN 事务处理

IN 事务处理用于实现设备到主机方向的数据传输，一个完整正确的 IN 事务处理如图 12-11 所示。



图 12-11　完整的 IN 事务处理

在正常的数据传输情况下，设备响应主机的 IN 令牌包，并向主机返回数据。传输过程中难免会出现各种错误，当发生如下情况时，将采用不同的处理方式。

情况 1：如果主机向设备发送的 IN 令牌包在传输过程中出现错误，设备接收不到正确的 IN 令牌包，此时设备将忽略该 IN 令牌包，不对该令牌包进行应答。

情况 2：如果设备接收到正确的 IN 令牌包，但是设备的 IN 端点被挂起（Suspend），无法向主机发送数据，此时设备将向主机发送 STALL 握手包。

情况 3：如果设备接收到正确的 IN 令牌包，但是设备由于某种原因无法向 USB 主机提供数据，此时设备将向主机发送 NAK 握手包。

情况 4：如果设备接收到正确的 IN 令牌包，并且向主机发送了数据包，但是数据包在传输过程中出现错误，主机接收不到正确的数据包，主机将忽略该错误的数据包，不做响应。

3）OUT 事务处理

OUT 事务处理用于实现主机到设备方向的数据传输，一个完整正确的 OUT 事务处理流程如图 12-12 所示。



图 12-12　完整的 OUT 事务处理

在正常的数据传输情况下设备响应主机的 OUT 令牌包，并接收主机发送的数据。在传输过程中发生以下情况，将采用不同的处理方式。

情况 1：如果主机向设备发送的 OUT 令牌包在传输过程中出现错误，设备接收不到正确的 OUT 令牌包，此时设备将忽略该令牌包，不对该令牌包进行应答。

情况 2：如果设备接收到正确的 OUT 令牌包，并且主机向设备发送数据包，但是在传输的过程中数据包出现错误，设备接收不到正确的数据包，此时设备将忽略该出错的数据包，不做响应。

情况 3：如果设备接收到正确的 OUT 令牌包，但是设备的 OUT 端点被挂起，无法接收

主机发来的数据,此时设备将向主机发送STALL握手包。

情况4:如果设备接收到正确的OUT令牌包,但是设备由于某种原因无法接收主机发来的数据,此时设备将向主机发送NAK握手包。

情况5:如果设备的数据触发位和接收到的数据包的触发位不一致,设备将丢弃该数据包,然后向主机发送ACK握手包。

5. USB数据传输类型

USB设备的每个有效端点与主机之间建立一个通道,每个通道以一种方式进行传输,传输分为多个阶段,分别由不同的事务构成。USB协议定义了4种传输类型,分别是批量传输、中断传输、同步传输和控制传输。以下将逐一介绍这4种传输方式,并对控制传输进行了详细的介绍。

1) 批量传输

该类型的传输用于传送大量的数据,要求传输不能出错,但是对时间没有要求,适用于打印机、扫描仪和存储设备等。在数据传输的过程中,当USB总线带宽紧张的时候,会自动为其他传输类型让出自己所占用的带宽,以降低自身的传输速率;当USB总线空闲时,就会占用较多的带宽,以提高自身的传输速率。批量传输可以发送大量的数据而不会阻塞USB总线,但其传输时间和传输速率得不到保障。在USB协议中,采用差错控制和重试机制来保证数据传输的正确性和可靠性。

2) 中断传输

中断传输适用于那些只传输少量数据并且请求传输的频率不高的一类设备。USB总线为中断传输保留了总线带宽,以保证它能在规定的时间内完成。

3) 同步传输

同步传输用于传输大量的、速率恒定的且对周期有要求的数据,适合于音频和视频类设备,因为这类设备需要数据的及时发送和接收,而对数据的正确性要求不是很高。USB协议为同步传输保留了总线带宽,以保证其一直使用准确的传输速率,因此传输时间是确定的和可预测的。此外,为了确保数据传输的及时性,同步传输没有采用差错控制和重试机制,即不能保证每次传输都是成功的。

4) 控制传输

控制传输是USB传输中最重要的传输类型,只有在正确执行完控制传输后,才能执行其他的传输类型。控制传输适用于低速、全速或高速的设备,主要传输少量的、对传输时间和传输速率没有要求,但必须保证传输完成的数据。控制传输用于USB主机和USB设备之间的配置、命令和状态通信流。所有的USB设备都支持控制传输,并且都有一个默认的地址0和控制端点0。当设备连接到主机时,首先进行控制传输并使用地址0和端点0用于交换信息,包括读取设备的描述符和设置设备的地址等。

控制传输一般包含两个或三个阶段:设置阶段、数据阶段(无数据控制没有此阶段)和状态阶段。

(1) 设置阶段。设置阶段是控制传输的第一阶段,由主机发起并向设备发送请求信息。该阶段包含一个SETUP令牌包、紧随其后的DATA0数据包(包内数据就是USB设备请求,固定为8字节)以及ACK握手包。该阶段是一次SETUP事务处理,定义了此次控制传输的内容,包括数据阶段的数据传输方向和要传输的数据的长度等。

（2）数据阶段。数据阶段用来传输主机与设备之间的数据。控制传输又可以分为三种类型：控制读取（读取设备描述符）、控制写入（配置设备）和无数据控制。控制读取时将数据从设备读到主机，读取的数据包括设备描述符等内容。该过程先由主机向设备发送 IN 令牌包，表明接下来主机要从设备读取数据。然后，设备向主机发送 DATA1 数据包。最后，主机将以下列方式加以响应：若数据正确被接收，主机发送 ACK 握手包；若主机正在忙碌没有接收到，主机发送 NAK 握手包；若传输发生错误，主机发送 STALL 握手包。控制写入则是将数据从主机传到设备上，所传的数据是对设备的配置信息。在该过程中，主机首先发送一个 OUT 令牌包，表明主机将要把数据发送到设备。接着，主机将数据通过 DATA1 数据包发送给设备。最后，设备将以下列方式加以响应：若数据被正确接收，设备发送 ACK 握手包；若设备正在忙碌没有接收到，设备发送 NAK 握手包；若传输发生错误，设备发送 STALL 握手包。如果数据阶段所传输的数据超过一个端点所支持的最大数据包长度，则将数据分布在多个事务处理中。DATA1 数据包和 DATA0 数据包需交替使用，且第一个数据包必须以 DATA1 数据包开始。

（3）状态阶段。状态阶段用来表示整个控制传输的过程已经结束了。需要注意的是，状态阶段传输的方向必须与数据阶段的方向相反。即：若数据阶段使用的是 IN 事务处理，则状态阶段应为 OUT 事务处理；若数据阶段使用的是 OUT 事务处理，则状态阶段应是 IN 事务处理。对于控制读取而言，此阶段主机会发送一个 OUT 令牌包，其后跟着一个无数据的 DATA1 数据包。此时设备会根据接收的情况向主机发送 ACK 握手包、NAK 握手包或者 STALL 握手包。相对地，对于控制写入而言，此阶段主机会发送一个 IN 令牌包，然后设备回发一个无数据的 DATA1 数据包，主机根据接收情况向设备发送 ACK 握手包、NAK 握手包或者 STALL 握手包。

需要注意的是，控制传输的前两个阶段——设置阶段和数据阶段的事务处理都会产生相应的中断，但是状态阶段的事务处理不会有中断产生。图 12-13 是一次控制传输的各个阶段，数据阶段是控制读取并且包含两个 IN 事务处理，因此状态阶段是一个 OUT 事务处理。

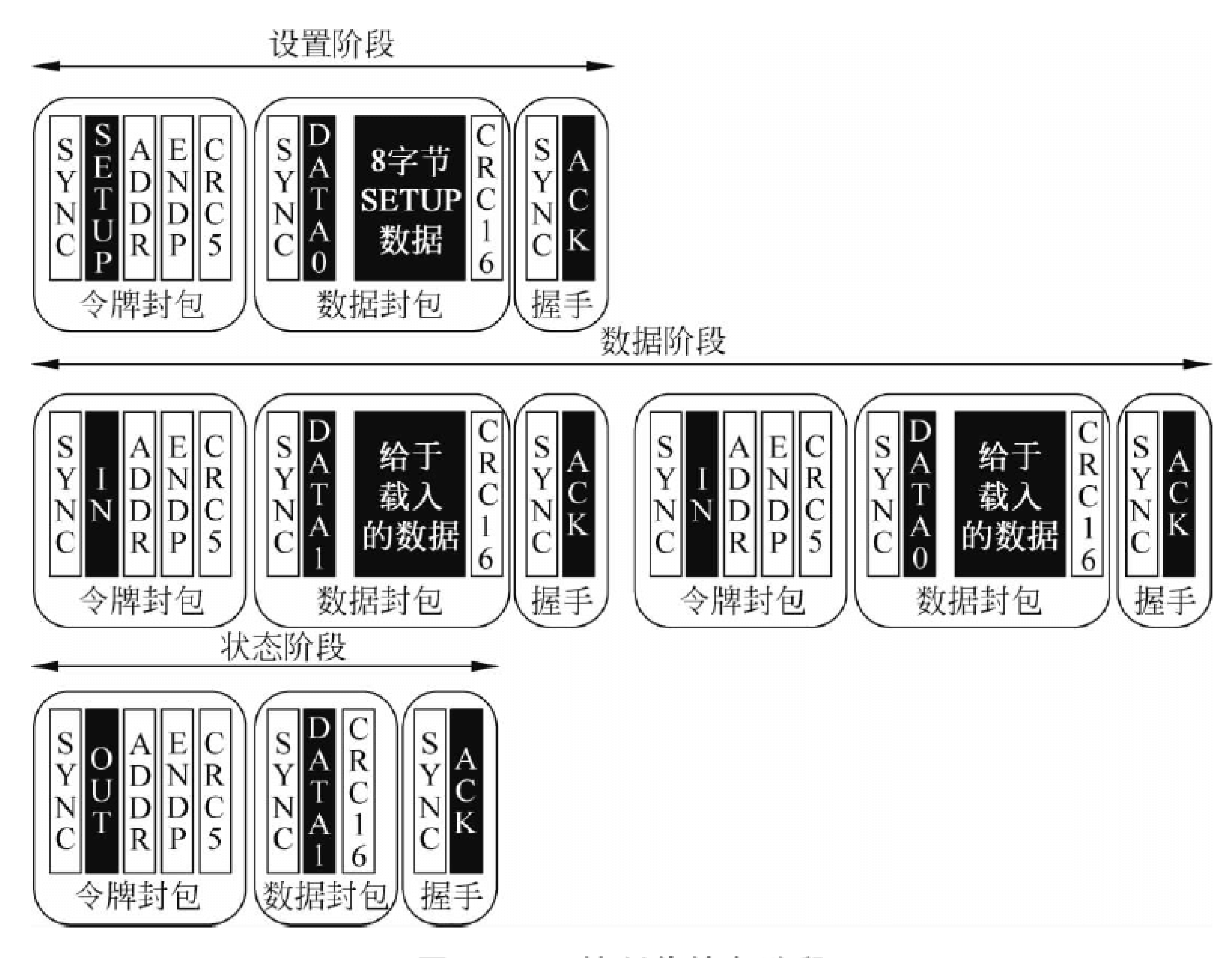


图 12-13　控制传输各阶段

12.4.2 USB底层编程涉及的描述符及设备请求

1. USB底层编程所涉及的描述符

可以认为USB设备是由一些配置、接口和端点等组件构成的,即一个USB设备可以含有一个或多个配置,在每个配置中可含有一个或多个接口,在每个接口中可含有若干个端点。其中,配置和接口是对USB设备功能的抽象,实际的数据传输是由端点来完成的。

USB设备使用各种描述符来说明整个设备或设备中某个组件的信息。描述符是一种数据结构,通常被保存在USB设备的固件程序中,使主机了解设备的格式化信息。描述符包括以下几种类型:设备描述符、配置描述符、接口描述符和端点描述符。这4种描述符是必须具有的;而其他的描述符,如字符串描述符、设备限定描述符以及其他速率配置描述符则是可选的。

下面通过举例,说明各描述符的结构和各字段含义。

1) 设备描述符

设备描述符用于说明设备的总体信息,包括描述设备的类型、设备支持的协议类型、供应商ID(Vender ID,VID)、产品ID(Product ID,PID)、设备版本号、供应商名称、产品名称及设备所支持的配置数等,目的是让主机获取插入的USB设备的属性,以便加载合适的驱动程序。一个USB设备只能有一个设备描述符,固定为18字节的长度,它是主机向设备请求的第一个描述符。

```
//设备描述符
const uint8 Device_Descriptor[18]=
{
    0x12,          //bLength 域,描述符的长度:18 字节
    0x01,          //bDescriptorType 域,描述符类型:0x01(表示本描述符为设备描述符)
    0x00,0x02,     //bcdUSB 域,USB 规范版本号(采用 BCD 码):2.0
    0x02,          //bDeviceClass 域,设备类代码
    0x00,          //bDeviceSubClass 域,设备子类代码
    0x00,          //bDeviceProtocol 域,设备协议代码(0x00 表示不使用任何设备类协议)
    0x20,          //bMaxPacketSize0 域,端点 0 支持最大数据包的长度:32 字节
    0xA2,0x15,     //idVendor 域,供应商 ID(VID)
    0x0F,0xA5,     //idProduct 域,产品 ID(PID)
    0x00,0x00,     //bcdDevice 域,设备版本号(采用 BCD 码)
    0x01,          //iManufacturer 域,供应商的字符串描述符索引:1
    0x02,          //iProduct 域,产品的字符串描述符索引:2
    0x03,          //iSerialNumber 域,设备序号的字符串描述符索引:3
    0x01           //bNumConfigurations 域,该 USB 设备支持的配置数目:1 个
};
```

2) 配置描述符

配置描述符用于说明USB设备中各个配置的特性,包括配置信息的总长度、配置所支持接口的个数、配置值、配置的属性及设备可以从总线提取的最大电流等。一个USB设备可以包含一个或多个配置(如USB设备的低速模式和高速模式可分别对应一个配置)。每一个配置都对应一个配置描述符,长度固定为9字节。

```
//配置描述符与
uin8 Configuration_Descriptor[9] =
{
0x09,       //bLength 域,配置描述符的长度: 9 字节
0x02,       //bDescriptorType 域: 0x02 表示本描述符为配置描述符
0x20,0x00,  //wTotalLength 域,配置信息的总长度(包括配置、接口和端点): 32 字节
0x01,       //bNumInterfaces 域,该配置所支持的接口数(至少一个): 1
0x01,       //bConfigurationValue 域,配置值: 1
0x04,       //iConfiguration 域,配置字符串描述符索引: 4
0x80,       //bmAttibutes 域,配置的属性(具有总线供电、自供电及过程唤醒的特性)
//位 7: 1-必须为 1,位 6: 0-自供电,位 5: 0-不支持远程唤醒
0x32        //MaxPower 域,设备从总线提取的最大电流以 2mA 为单位: 50×2mA=100mA
};
```

在使用 USB 设备前,主机必须为其选择一个合适的配置。主机根据设备描述符所支持的配置数按顺序查找所有的配置描述符,继而查找接口描述符以及端点描述符,直到查找到主机所支持的配置,需要注意的是,一个 USB 设备在一个时刻只能工作在一种配置下。

3) 接口描述符

接口描述符用于说明 USB 设备中各个接口的特性,包括接口号、接口使用的端点数、所属的设备类及其子类等。一个配置可以包含一个或多个接口,每个接口都必须有一个接口描述符。如对一个光驱来说,当用于文件传输时使用其大容量存储接口;当用于播放 CD 时,使用其音频接口。接口是端点的集合,可以包含一个或多个可替换设置,用户能够在 USB 处于配置状态时,改变当前接口所含的个数和特性。

```
//接口描述符
uint8 interface_descriptor[9] =
{
0x09,    //bLength 域,接口描述符长度: 9 字节
0x04,    //bDescriptorType 域: 0x04 表示本描述符为接口描述符
0x00,    //bInterfaceNumber 域,接口号
0x00,    //bAlternateSetting 域,接口的可替换设置值
0x02,    //bNumEndpoints 域,接口使用的端点数(除端点 0): 2
0xff,    //bInterfaceClass 域,接口所属的 USB 设备类: 0xFF 表示供应商自定义
0xff,    //bInterfaceSubClass 域,接口所属的 USB 设备子类: 0xFF 表示供应商自定义
0xff,    //bInterfaceProtocol 域,接口采用的 USB 设备类协议: 0xFF 表示供应商自定义
0x05     //iInterface 域,接口字符串描述符的索引: 5
};
```

4) 端点描述符

所有的传输都是传送到设备端点,或是从设备端点发出。这种端点实际上就是一个能够存储多个字节的缓冲器。端点通常是一个数据存储器区块,或是控制器芯片中的一个寄存器,端点所存储的数据可能是收到的数据,或是等待发送的数据。端点所支持的传输类型和传输方向等信息,都在端点描述符中定义,端点描述符长度为 7 字节。每个端点的描述符总是作为配置描述符的一部分返回,端点 0 无描述符,其他端点必须包含端点描述符。

```
//端点 2 描述符
uint8 endpoint_descriptor2[7] =
{
0x07,        //bLength 域,端点描述符长度: 7 字节
0x05,        //bDescriptorType 域: 0x05 表示本描述符为端点描述符
0x82,        //bEndpointAddress 域,端点号和传输方向: 端点 2、IN
0x02,        //bmAttributes 域,端点特性: 数据端点、块传输
0x00,0x02,   //wMaxPacketSize 域,端点支持最大数据包长度: 512 字节
0x00         //bInterval 域,轮询间隔,以 ms 为单位。
};
//端点 3 描述符
uint8 endpoint_descriptor3[7] =
{
0x07,        //bLength 域,端点描述符长度: 7 字节
0x05,        //bDescriptorType 域: 0x05 表示本描述符为端点描述符
0x03,        //bEndpointAddress 域,端点号和传输方向: 端点 3、OUT
0x02,        //bmAttributes 域,端点特性: 数据端点、块传输
0x00,0x02,   //wMaxPacketSize 域,端点支持最大数据包长度: 512 字节
0x00         //bInterval 域,轮询间隔,以 ms 为单位。
};
```

主机针对收到的数据和要发出的数据也有缓冲器,但是主机并没有端点,而是与设备各个端点进行通信的出发点。在USB 2.0协议中,设备端点被定义为“USB设备中的唯一可寻址的部分,是主机和设备之间通信流的来源和去向”。除了端点0外,其他端点只能在单方向上传输数据(上行传输或者下行传输)。端点方向是基于主机角度定义的:IN表示发送数据到主机(上行传输),OUT表示主机发送数据(下行传输)。主机在对USB设备配置时,也会分配给相应的逻辑设备一个唯一的地址。由设备地址、端点号和端点方向就可以唯一指定一个端点。

5) 字符串描述符

在USB设备中通常还含有字符串描述符,以说明一些专用信息,如制造商的名称、设备的序列号等。它的内容以UNICODE的形式给出,且可以被客户软件所读取。对USB设备来说,字符串描述符是可选的。

2. 缓冲区描述符表

USB使用缓冲区描述表(Buffer Descriptor Table,BDT)来高效管理USB端点通信。BDT中的每一个表项,即缓冲区描述符(Buffer Descriptor,BD)占8个字节(64位),包括32位的控制/状态字和32位的缓冲区地址。BD各字段分配如表12-6所示。每个端点的方向需要两个BD(EVEN BD和ODD BD)。这样,当微处理器处理其中一个BD时,USB模块就可以处理另一个,这种双缓冲方式能使数据传输达到最大吞吐量。一个接口最多有16个双向端点,因此BDT表最多占据系统存储空间512字节。

表12-6　缓冲区描述符(BD)的格式

D25～D16	D7	D6	D5	D4	D3	D2
BC	OWN	DATA0/1	KEEP/PID[3]	NINC/PID[2]	DTS/PID[1]	BDT_STALL/ PID[0]

需要注意的是，BDT 表的开始地址必须位于系统内存的 512 字节对齐的边界上，也就是说 BDT 表的开始地址的低 9 位全为 0。

缓冲区描述符 BD 提供了端点缓冲区控制信息。根据 BD 的控制权是 USB 模块还是 MCU，BD 有不同的含义。

USB 模块根据存储在 BD 中的数据来决定：

(1) 谁拥有缓冲区的操作权；

(2) DATA0 还是 DATA1 数据包；

(3) 根据包是否被完成以决定是否释放拥有者；

(4) 无地址增加(FIFO)；

(5) 数据同步机制是否被使能(DATA0/DATA1)；

(6) 有多少数据被发送和接收；

(7) 缓冲区是否位于内存。

MCU 根据存储在 BD 中的数据来决定：

(1) 谁拥有缓冲区操作权；

(2) DATA0 还是 DATA1 数据包；

(3) 有多少数据被发送和接收；

(4) 缓冲区是否位于内存。

D31～D26，D15～D8，D1～D0——保留。

D25～D16——接收到的数据字节数。SIE 会根据接收到的字节数改变该字段的值。

D7——决定当前是微处理器还是 USB 模块拥有对缓冲区描述符表的操作权。0：处理器；1：USB 模块。当 KEEP 位为 0 时，在完成一个 TOKEN 后向该位写 0。

D6——指明发送或接收了 DATA 0 还是 DATA 1 数据包。0：DATA 0 包，1：DATA 1 包。

D5——如果 KEEP 置 1 且 OWN 置 1，USB 模块将持久地对 BD 进行互斥访问。当 TOKEN 已经被处理时，KEEP 位应被清 0，此时 USB 模块才释放 BD。一般地，对于实时端点(送给 FIFO)，该位置 1。在这种情况下，NINC 通常置 1，禁止地址增加。如果 KEEP 置 1，OTG 模块不会改变该位；否则当前令牌 PID 的第三位被写回。

D4——NINC 置 1 时访问数据缓冲区，DMA 引擎将不会增加它的地址，这对于在一个存储器中存取数据的端点来说是很用的。

D3——DTS 被置 1，将使能 USB 模块的同步触发，如果 KEEP 置 1，OTG 模块不改变该位。

D2——如果将该位置 1，解释到一个 BD 处理的令牌，将使 OTG 模块发出一个 STALL 握手协议。在这种情况下，BD 的内容和它的 OWN 位不会改变。如果 KEEP 置 1，OTG 模块不会改变该位；否则当前令牌 PID 的第 0 位被写回。

当一个被使能的端点完成一个事务处理时，USB 模块会使用其集成的 DMA 控制器去访问 BDT，USB 模块去读相关端点的 BD 以决定其是否拥有 BD 和相关缓冲区的操作权。为了计算在 BDT 中的入口，BDT 页寄存器、当前端点、TX 和 ODD 字段构成一个 32 位的地址。该地址各字段分配如表 12-7 所示。

表 12-7　BDT 地址计算

31:24	23:16	15:9	8:5	4	3	2:0
BDT_PAGE_03	BDT_PAGE_02	BDT_PAGE_01	端点	IN	ODD	000

BDT_PAGE：BDT 页寄存器。

END_POINT：令牌包中的端点号。

TX：1：发送传输；0：接收传输。

ODD：由 USB 模块的 SIE 维护。ODD 与当前使用的缓冲区相关联。

3. 标准设备请求

所有的 USB 设备都应该能够对来自 USB 主机的请求做出响应。这些请求是在控制传输的 SETUP 阶段由主机发往设备的，并使用默认的控制管道，各个字段由主机定义，表达了一次控制传输的目的。对于标准设备请求，不管设备是否被分配了新的地址或者已被配置，设备都需要做出响应。

以主机获取设备的描述符为例，在控制传输的 SETUP 阶段，主机发往设备的数据包中会包含 GET_DESCRIPTOR(获取描述符)的请求，在数据阶段将能收到设备发来的相关描述符。

1) 设备请求字段格式

设备请求一共有 8 个字节，格式如表 12-8 所示。

表 12-8　设备请求格式

位移量	字段值	大小/B	描　　述
0	bmRequestType	1	D7：数据传输方向。0：主机至设备，1：设备至主机 D[6:5]：类型。0：标准，1：类，2：供应商，3：保留 D[4:0]：接收端。0：设备，1：接口，2：端点，3：其他 4～31：保留
1	bRequest	1	指定具体是什么请求
2	wValue	2	用于传递参数，长度根据请求的不同而不同
4	wIndex	2	用于传递参数，根据请求的不同而不同，通常是传递索引和位移量
6	wLength	2	如果有数据阶段，该域表示所要传输的字节大小

(1) bmRequestType。该字段定义了指定请求的一些特性，比如控制传输的第二阶段数据传输的方向，如果 wLength 字段置为 0，表明没有数据阶段，那么该字段的数据传输方向位会被忽略。设备请求的接收端可以是设备、接口或者端点等，当是接口或者端点时，wIndex 字段用于指定该接口或者端点。

(2) bRequest。该字段指明了具体是什么设备请求，bmRequestType 字段的类型位可修改该字段含义。仅当类型位被置 0 时(表示是标准设备请求)，该字段的值才有定义。表 12-9 给出了该字段的取值对应的标准设备请求。

(3) wValue。该字段的值根据设备请求的不同而不同，用于向设备传递参数。当设备请求是标准设备请求中的获取描述符时(bRequest 的值为 GET_DESCRIPTOR)，wValue 字段的值表明了所要获取的描述符的类型。表 12-10 是该字段的取值对应的描述符类型。

表 12-9　bRequest 的取值

请　　求	值	描　　述
GET_STATUS	0	获取指定接收者的状态
GET_FEATURE	1	获取指定的特性
保留	2	为将来使用
SET_FEATURE	3	设置指定的特性
保留	4	为将来使用
SET_ADDRESS	5	设置地址
GET_DESCRIPTOR	6	获取描述符
SET_DESCRIPTOR	7	更新描述符
GET_CONFIGURATION	8	获取配置
SET_CONFIGURATION	9	设置配置
GET_INTERFACE	10	获取接口
SET_INTERFACE	11	设置接口
SYNCH_FRAME	12	同步帧

表 12-10　wValue 的取值

描述符类型	值	描　　述
DEVICE	1	设备描述符
CONFIGURATION	2	配置描述符
STRING	3	字符串描述符
INTERFACE	4	接口描述符
ENDPOINT	5	端点描述符
DEVICE_QUALIFIER	6	设备限定描述符
OTHER_SPEED_CONFIGURATION	7	其他速率配置描述符
INTERFACE_POWER	8	接口能量描述符

（4）wIndex。该字段的值根据设备请求的不同而不同，用于向设备传递参数，常用于指定接口或者端点。当用于指定端点时，D[0:3]表示端点号，D[4:6]保留，D[7]表示该端点的方向，D[8: 15]保留；当用于指定接口时，D[0:7]表示接口号，D[8:15]保留。

（5）wLength。该字段指定了控制传输第二阶段需传输数据的长度，当该字段的值为 0 时，则没有数据阶段。在输入请求下，要求设备返回的数据长度一定不能大于 wLength 指定的长度，但是可以少于；在输出请求下，wLength 指定了主机要发送的确切数据长度。

2）各种类型的设备请求

（1）获取描述符请求。获取描述符请求（Get）用于 USB 主机读取 USB 设备指定的描述符，在该请求的数据阶段，USB 设备将向 USB 主机返回指定的描述符。该请求的各个字段为：

bmRequestType	bRequest	wValue	wIndex	wLength
10000000B	GET_DESCRIPTOR	描述符类型/索引	0 或语言 ID	描述符长度

(2) 设置描述符请求。设置描述符请求是可选的，用于更新已经存在的描述符或者添加新的描述符。在该请求的数据阶段，USB主机将向USB设备发送指定的描述符数据。该请求的各个字段为：

bmRequestType	bRequest	wValue	wIndex	wLength
00000000B	SET_DESCRIPTOR	描述符类型/索引	0或语言ID	描述符长度

(3) 设置配置请求。设置配置请求与获取配置请求相对应，该请求用于为USB设备设置一个合适的配置值，且无数据阶段。配置值wValue必须是0或者是与配置描述符相匹配的配置值。该请求的各个字段为：

bmRequestType	bRequest	wValue	wIndex	wLength
00000000B	SET_CONFIGURATION	配置值	0	1

(4) 获取配置请求。获取配置请求主要用于USB主机读取USB设备当前的配置值，在该请求的数据阶段，USB设备将向USB主机返回一个字节的配置值。该请求的各个字段为：

bmRequestType	bRequest	wValue	wIndex	wLength
10000000B	GET_CONFIGURATION	0	0	1

在USB设备处于不同状态时，该请求具有不同的响应：当USB设备处于地址状态时，该请求返回0；当USB设备处于配置状态时，该请求返回当前配置描述符中的bConfigurationValue字段的值；USB设备处于默认状态时，该请求无效。

(5) 获取接口请求。获取接口请求主要用于USB主机读取指定接口的可替换设置值，也就是接口描述符中的bAlternateSetting字段的值。在获取接口请求的数据阶段，USB设备将向USB主机返回一个字节的可替换设置值。该请求的各个字段为：

bmRequestType	bRequest	wValue	wIndex	wLength
10000001B	GET_INTERFACE	0	接口	1

(6) 设置接口请求。设置接口请求和获取接口请求是相对应的，该请求用于为指定的接口选择一个合适的可替换设置值，该请求没有数据阶段。另外，该请求只在USB设备处于配置状态时有效。当USB设备的一个接口存在一个可替换设置时，该请求使得主机可以为其选择所需要的可替换设置。该请求的各个字段为：

bmRequestType	bRequest	wValue	wIndex	wLength
10000001B	SET_INTERFACE	0	接口	1

(7) 设置地址请求。设置地址请求主要用于在USB设备上电的时候，为其分配一个唯一的设备地址，该请求没有数据阶段，状态阶段的方向则是设备到主机，其中，wVlaule字段

为新的设备地址。该请求的各个字段为：

bmRequestType	bRequest	wValue	wIndex	wLength
00000000B	SET_ADDRESS	设备地址	0	0

(8) 获取状态请求。获取状态请求主要用于 USB 主机读取 USB 设备、接口或端点的当前状态。在该请求的数据阶段，USB 设备将向 USB 主机返回具有特定格式的两个字节数据。如果 wValue 或 wLength 字段没有被赋值，或者在获取 USB 设备状态请求中的 wIndex 字段非 0，则 USB 设备的行为是不确定的。如果 USB 设备中不存在该请求中指定的接口或者端点，USB 设备将向 USB 主机返回一个错误。下面将分别介绍该请求用于获取设备、接口或端点状态的情况。该请求的各个字段为：

bmRequestType	bRequest	wValue	wIndex	wLength
10000000B	GET_STATUS	0	0	2
10000001B			接口	
10000010B			端点	

读取设备：获取设备状态的请求返回给主机的数据，D15～D2 保留，只有 D1、D0 有意义。D0 位表示设备是否为自供电，如果该位为 0，则表示设备是总线供电；如果该位是 1，则表示设备是自供电。D1 位表示设备是否支持远程唤醒(设备一般默认不支持远程唤醒)，如果该位是 0，则表示该功能被禁止；如果该位是 1，则表示该功能被启用。

读取接口：获取接口状态的请求返回给主机的数据，两个字节数据 D15～D0 全部保留。

读取端点：获取端点状态的请求返回给主机的数据，D15～D1 保留，只有 D0 有意义。D0 位表示端点的停止特性；如果该位是 0，则表示指定的端点未被停止；如果该位是 1，则表示指定的端点已被停止。

12.4.3　USB 设备状态

设备有多种可能的状态，有些状态对于主机来说是可见的，而有些状态是设备内部的。USB 设备状态有以下几种。

(1) 连接状态。当 USB 设备通过 USB 电缆连接到 USB 主机或 Hub 的下行端口时，即进入连接状态，此时 USB 总线开始向 USB 设备供电，至电源稳定工作。

(2) 上电状态。当连接的 USB 主机的 USB 设备得到稳定的 USB 总线电源后，便处于上电状态，此时设备还没有被复位，不能对任何事务进行处理。设备可以从 USB 总线上获取电源，也可以自供电。

(3) 默认状态。在 USB 设备上电后，USB 设备会响应 USB 主机发出的复位信号，进行复位操作。复位结束后，USB 设备进入默认状态。在默认状态下，USB 设备可以从 USB 总线上获得不超过 100mA 的电流，并使用默认的设备地址 0 进行事务处理。

(4) 地址状态。USB 设备复位结束后，USB 主机重新为设备分配一个地址，这个地址是唯一的，此时设备便处于地址状态。在地址状态后，设备将不能使用默认的设备地址，而

使用新分配的地址来进行之后的总线活动。

(5) 配置状态。在 USB 设备被使用之前，设备必须先被配置。主机发出一个带有非零配置值的配置请求(SET_CONFIGURATION)，设备在正确处理配置请求后，便进入配置状态。在配置状态下，所有的寄存器返回至默认值。

(6) 挂起状态。USB 协议规定，如果 USB 设备在 3ms 内没有检测到总线活动，将自动进入挂起状态，这样可以节省功耗。上电后的 USB 设备无论是否被分配一个新的地址或是否被配置，都必须时刻准备进入挂起状态。

12.4.4　USB 总线的枚举过程

(1) 当设备插入到 Hub 端口时，有上拉电阻的一根数据线被拉高到幅值的 90%的电压(大约是 3V)。Hub 检测到它的一根数据线是高电平，就认为有设备插入，并能根据是 D+ 还是 D− 被拉高来判断是低速、全速还是高速设备。检测到设备后，Hub 继续给设备供电，但并不急于与设备进行通信。

(2) 每个 Hub 利用它自己的中断端点向主机报告它各个端口的状态(这个过程对于设备而言是看不到的，也不必关心)，报告的内容只是 Hub 端口的设备连接/断开事件。如果有连接/断开事件发生，那么主机会向 Hub 发送一个 Get_Port_Status 请求(Get_Port_Status 请求是所有 Hub 都支持的标准请求)以了解此次端口状态改变的确切含义。Hub 收到该请求后，会将插入到该端口的设备的速度信息(低速、全速、高速)返回给主机。

(3) 主机得知有新设备连接上后，主机至少需要等待 100ms 以使插入操作完成和设备电源稳定。然后 USB 主机控制器向 Hub 发送一个 Set_Port_Feature 请求让 Hub 复位该端口(刚才设备插入的端口)。Hub 通过驱动设备的数据线到复位状态(D+ 和 D− 全是低电平)，并持续至少 10ms。Hub 并不会复位其他已有设备连接的端口，所以连接在该 Hub 上的其他设备并不会受到影响。

(4) 根据 USB 2.0 协议，高速设备在开始时默认使用全速模式，所以对于一个支持 USB2.0 的高速 Hub，当它发现其端口连接的是一个全速设备时，会进行高速检测，看看目前这个设备是否还支持高速模式，如果是，那就切换到高速模式，否则就一直在全速模式下运行。从设备的角度来看，如果是一个高速设备，在刚连接 Hub 或上电时只能用全速模式。随后 Hub 会进行高速检测，之后这个设备才会切换到高速模式下工作。假如 Hub 不支持 USB 2.0(不支持高速模式)，那么将不能进行高速检测，设备将一直以全速模式工作。

(5) 主机不停地向 Hub 发送 Get_Port_Status 请求，以查询设备是否复位成功。Hub 返回的报告信息有一位专门用来标志设备的复位状态。当设备复位成功后，Hub 将撤销复位信号，设备便处于默认状态，并使用默认地址 0 和端点 0 来接收主机发来的请求。此时，设备能从总线上获得的最大电流是 100mA。

(6) 主机给设备发送获取设备描述符请求(Get_DESCRIPTOR)，设备返回 18 字节的设备描述符。这是主机第一次得到设备描述符，主机并不会分析各个字段的含义，只会设备描述符中端点 0 所支持的最大数据包长度(设备描述符的第 8 字节)。当控制传输的状态阶段完成后，主机会要求 Hub 再对设备进行一次复位操作。

(7) USB 主机控制器通过 Set_Address 请求向设备分配一个唯一的地址。在完成这次控制传输后，设备将进入地址状态，之后将使用新的地址继续与主机通信。这个新的地址对

于设备来说是终身制的，只要设备不被拔出、复位或者系统重启，那么该地址将一直存在。另外，同一个设备再次被枚举得到的地址就不一定和上一个地址相同了。

(8) 主机再一次向设备发送获取设备描述符请求，这次主机将会认真解析设备描述符的内容，包括端点 0 支持的最大数据包长度、设备所支持的配置数、供应商 ID(VID)、产品 ID(PID)等信息。之后主机发送获取配置描述符请求，主机先获得 9 字节的配置描述符，其中包括配置描述符和其下层的所有描述符的总长度。接着主机再一次发送获取配置描述符请求，这一次设备会把配置描述符、接口描述符和端点描述符一并发给主机，主机则根据每个描述符的 bDescriptortype 字段来判断收到的具体是哪种描述符。如果还有字符串描述符，主机还要继续获取。另外，如果是 HID 设备，还应有 HID 描述符。

(9) 此时主机会弹出窗体，展现发现的新设备的信息。主机通过解析获得的描述符以对设备有足够的了解，会选择一个最合适的驱动程序给设备，并将设备添加到 USB 总线的设备列表中，并把对设备的控制权交给驱动程序。

(10) 驱动程序会要求设备重新发送描述符，并为设备选择一个合适的配置值。在通过描述符获悉设备的状况后，驱动程序向设备发送一个带有所需配置值的 Set_Configuration 请求。设备接收到请求后，使能所要求的配置。这就使设备处于配置状态，此时设备可以进行数据传输。

12.5　KL25/26 芯片 USB 模块的编程结构

12.5.1　USB 模块寄存器

KL25/26 的 USB 模块主要使用到表 12-11 中的寄存器，其他寄存器的详细信息，读者可以参考芯片手册内容。

表 12-11　USB 模块寄存器列表

寄　存　器	缩　　写	描　　述
中断状态寄存器	USBx_ISTAT	USB 模块中断源
中断使能寄存器	INT_ENB	USB 模块中断使能
状态寄存器	STAT	记录 USB 模块中事务状态
控制寄存器	CTL	提供 USB 模块各种控制和配置
地址寄存器	ADDR	根据主从不同而定
BDT 页寄存器 1	BDT_PAGE_01	页寄存器 1
Token 寄存器	TOKEN	主机模式下发起事务
BDT 页寄存器 2	BDT_PAGE_02	页寄存器 2
BDT 页寄存器 3	BDT_PAGE_03	页寄存器 3
端点控制寄存器 0 …… 端点控制寄存器 15	ENDPT0 …… ENDPT15	对各个端点的控制

1. 中断状态寄存器

中断状态寄存器(Interrupt Status Register，USBx_ISTAT)为 8 位寄存器，与中断使能

寄存器(USBx_INTEN)的8位一一对应。中断位为1后,通过向中断位写1清零,其结构如表12-12所示。复位后该寄存器的值为0x00。

表12-12　USBx_ISTAT结构

数据位	D7	D6	D5	D4	D3	D2	D1	D0
读	STALL	ATTACH	RESUME	SLEEP	TOKDNE	SOFTOK	ERROR	USBRST

在程序开发中主要考虑D6、D3和D0位。

D6——设备连接位。当USB模块检测到USB设备连接时该位置1。

D3——令牌完成位。当处理完当前的令牌后,USB模块自动将该位置1。软件向该位写1将清0。

D0——USB复位位。USB模块解析到有效的USB复位(2.5ms)就将该位置1,从而通知微处理器向地址寄存器中写入0x00并使能端点0。

2. 中断使能寄存器

中断使能寄存器(Interrupt Enable Register,USBx_INTEN)的每一位控制USB模块中的一个中断源,其结构如表12-13所示。置1使能INT_STAT寄存器中相应的中断源。复位后该寄存器的值为0x00。

表12-13　USBx_INTEN结构

数据位	D7	D6	D5	D4	D3	D2	D1	D0
读/写	STALLEN	ATTACHEN	RESUMEEN	SLEEPEN	TOKDNEEN	SOFTOKEN	ERROREN	USBRSTEN

与中断状态寄存器一样,在程序开发中主要考虑D3和D0位。

D3——令牌完成使能位。0:禁止,1:使能。

D0——USB复位使能位。0:禁止,1:使能。

3. 状态寄存器

状态寄存器(Status Register,USBx_STAT)记录了USB模块中事务的状态,其结构如表12-14所示。当产生令牌完成中断时,应先读取状态寄存器以确定端点的通信状态。复位后该寄存器的值为0x00。

表12-14　USBx_STAT结构

数据位	D7	D6	D5	D4	D3	D2	D1	D0
读	ENDP				TX	ODD	0	

D7~D4——端点号。该4位值对应令牌完成中断对应的端点号,从而确定对应的BDT选项。

D3——传输方向指示位。0:上一个事务为接收操作;1:上一个事务为发送操作。

D2——如果最后更改的缓冲区描述符位于BDT的奇数行就将该位置1。

4. 控制寄存器

控制寄存器(Control Register,USBx_CTL)提供了USB模块的各种控制和配置信息,其结构如表12-15所示。复位后该寄存器的值为0x00。

表 12-15　USBx_CTL 结构

数据位	D7	D6	D5	D4	D3	D2	D1	D0
读/写	JSTATE	SE0	TXSUSPENDTOKENBUSY	RESET	HOSTMODEEN	RESUME	ODDRST	USBENSOFEN

D7——USB 差分接收 J 状态信号。该信号的极性受地址寄存器 USBx_ADDR 的 LSEN 位状态的影响。

D6——USB 的 SE0 信号。

D5——在主机模式时，将该位置 1 后，USB 模块将处理 USB 令牌，此时不能再向令牌寄存器写入令牌命令。从机模式时，SIE 禁止了包发送和接收时，将 TXSUSPEND 置 1。清 0 该位，SIE 能继续处理令牌。接收到 Setup 令牌后 SIE 将该位置 1。

D4——将该位置 1 使能 USB 模块产生复位信号，从而使能 USB 模块复位 USB 外设。该信号只有在主机模式下（HOSTMODEEN＝1）才有效，在所需时间内，软件必须将 RESET 置 1 然后清 0。

D3——主机模式使能位，为 1 使能 USB 模块工作于主机模式。

D2——将该位设为 1 使能 USB 模块产生唤醒信号，从而使能 USB 模块远程唤醒。软件必须先将 RESUME 位置 1，等待规定的时间后再清 0。如果 HOSTMODEEN 位为 1，当将 RESUME 位清 0 时，USB 模块在唤醒信号后添加一个低速结束包。

D1——将该位置 1 会将 BDT 的所有位复位为 0，同时指定到 BDT 的偶数行。

D0——USB 使能位。0：禁止 USB 模块，1：使能 USB 模块。

5．地址寄存器

作为外设模式（HOSTMODEEN＝0），USB 模块解码时，地址寄存器（Address Register，USBx_ADDR）有唯一值。而作为主机模式（HOSTMODEEN＝1），USB 模块使用令牌包传输地址，从而使能 USB 模块为 USB 外设指定唯一地址。无论哪种模式，控制寄存器的 USBENSOFEN 位均为 1，其结构如表 12-16 所示。复位或 USB 模块检测到复位信号时，地址寄存器复位为 0。复位后该寄存器的值为 0x00。

表 12-16　USBx_ADDR 结构

数据位	D7	D6	D5	D4	D3	D2	D1	D0
读/写	LSEN	ADDR						

D7——低速使能位。该位通知 USB 模块写入令牌寄存器的下一个令牌命令时，必须以低速执行。

D6～D0——USB 地址。定义了在外设模式下，USB 模块解析的外设地址，或主机模式下传输的地址。

6．BDT 页寄存器

缓冲区描述符表（BDT）页寄存器 1～3 用来计算当前缓冲区描述符表在系统存储空间的地址。

其中，8 位 USBx_BDTPAGE3 寄存器对应 BDT_BA31～BDT_BA24、8 位 USBx_BDTPAGE2 寄存器对应 BDT_BA23～BDT_BA16、8 位 USBx_BDTPAGE1 寄存器 D7～D1 对应 BDT_BA15～BDT_BA9，D0 保留。

事实上,缓冲区描述符表BDT的32位地址通过如表12-17所示方式拼接而成。

表12-17　BDT结构

D31～D24	D23～D16	D15～D9	D8～D5	D4	D3	D2～D0
USBx_ BDTPAGE3	USBx_ BDTPAGE2	USBx_BDTPAGE1 [7:1]	ENDP	TX	ODD	0 0 0

其中,ENDP、TX以及ODD为状态寄存器STAT内容,分别表示端点号、传输方向以及DATA0或DATA1。

7. 令牌寄存器

令牌寄存器(Token Register,USBx_TOKEN)用于在主机模式下(HOSTMODEEN=1)执行USB事务,其结构如表12-18所示。当处理器要向外设发起一个事务时,将令牌类型和端点写入该寄存器。写完后,USB模块开始向地址寄存器中的地址发起指定的事务。复位为0。

表12-18　USBx_TOKEN结构

数据位	D7	D6	D5	D4	D3	D2	D1	D0
读/写	TOKENPID				TOKENENDPT			

D7～D4——USB模块执行的令牌类型。有效的令牌类型有0001(OUT令牌)、1001(IN令牌)、1101(SETUP令牌)等。

D3～D0——令牌命令的端点地址。

8. 端点控制寄存器

USB模块中有16个可用端点,端点控制寄存器(Endpoint Control Registers,ENDPT0～15)拥有每个端点的控制位,如表12-19所示。复位为0。

表12-19　ENDPT0～15结构

数据位	D7	D6	D5	D4	D3	D2	D1	D0
读/写	HOSTWOHUB	RETRYDIS	0	EPCTLDIS	EPRXEN	EPTXEN	EPSTALL	EPHSHK

D7——该位只能用于主机模式并且只在端点0(ENDPT0)控制寄存器中有效。该位为1时允许主机直接和低速设备通信。为0时,主机需要通过HUB与低速设备通信,发送令牌给低速设备时,主机提供PRE_PID然后切换到低速信号。

D6——该位只能用于主机模式并且只在端点0(ENDPT0)控制寄存器中有效。将该位设为1时,主机将重发NAK(不响应)事务。事务没有响应时,用NAK PID更新BDT PID,并将TOKENDNE中断置1。

D5——该位一直为0。

D4——EPCTLDIS:该位置1会禁止控制传输(SETUP),设为0时使能控制传输。只有当EPRXEN和EPTXEN位都为1时才有效。

D3——该位为1时,使能端点为RX传输。

D2——该位为1时,使能端点为TX传输。

D1——该位为 1 时表明使用了该端点，该位的优先级比端点使能寄存器的其他位都高，只有在 EPTXEN=1 或 EPRXEN=1 时才有效。访问该端点会使得 USB 模块返回一个 STALL 握手包。

D0——该位设为 1 时，该端点的事务期间能执行握手。

表 12-20 给出了端点寄存器的 EPCTLDIS 位、EPRXEN 位以及 EPTXEN 位的设置示例及其含义。

表 12-20　端点方向和控制

EPLCTLDIS	EPRXEN	EPTXEN	端点使能/方向控制
X	0	0	禁止端点
X	0	1	端点只允许 TX 传输
X	1	0	端点只允许 RX 传输
1	1	1	端点允许 TX 和 RX 传输
0	1	1	端点允许 TX 和 RX 传输及控制(SETUP)传输

12.5.2　USB 模块中断详解

USB 设备通信时基本采用中断方式(此处的中断方式与中断传输方式说的是不同的方面)，具体用到的中断类型有复位中断、ERROR 中断、SOF 中断、令牌完成中断、SLEEP 中断、RESUME 中断、ATTACH 中断、STALL 中断，以下将对各个中断进行详细的介绍。

1. 复位中断

USB 设备连接上 USB 主机后，USB 主机给 USB 设备上电，USB 设备上电完成之后，USB 主机会给 USB 设备发送一个复位信号对 USB 设备进行复位。USB 设备接收到复位信号之后，就是产生一个复位中断，从而执行相应的复位中断处理函数。在复位中断处理函数中，USB 设备会清除所有中断标志，并且安装默认的端点 0 与 USB 主机进行通信，执行后续的枚举。

2. ERROR 中断

在 USB 通信过程中，如果发生错误，则会产生一个 ERROR 中断。这些错误可能包括总线超时错误、CRC16 错误和 PID 检查错误等。

3. SOF 中断

USB 通信协议中规定，USB 主机是在一个帧内发送数据包(Package)的，USB 主机会提前发送一个 SOF 包，表明是一个帧的开始。USB 设备接收到 SOF 包之后，会向 USB 主机发送一个确认包，然后会产生 SOF 中断，在 SOF 中断复位例程中清 SOF 中断标志位。

4. 令牌完成中断

令牌完成中断完成枚举过程和数据的传输。在 USB 主机枚举 USB 设备时，USB 主机会向 USB 设备发送相关设备请求(具体见 USB 基本知识要素)，这些请求包含在 SETUP 包中，在这次 SETUP 事务处理成功完成后，USB 设备会产生一个 SETUP 令牌中断。类似地，在进行 USB 主机和 USB 设备之间的数据传输时，USB 设备同样会产生令牌中断。如果 USB 主机向 USB 设备发送数据，则 USB 设备会产生一个 OUT 令牌中断，并调用接收数据函数接收数据；如果 USB 主机向 USB 设备要数据，则 USB 设备会产生 IN 令牌中断，并

调用发送数据函数发送数据。

5. SLEEP 中断

当 USB 主机在一段时间内检测到 USB 总线上没有相应的设备在活动时，USB 主机会向 USB 设备发送一个 SLEEP 信号，让相应的 USB 设备进入睡眠状态以节省功耗。本测试例程中，SLEEP 中断服务例程中只清除中断标志位。

6. RESUME 中断

将该位置 1 决定于 DP/DM 信号，并且可以被用来当作远程唤醒 USB 总线的信号。当不是处于休眠状态时，该位应该被禁止。

7. ATTACH 中断

这一中断用在主机中，如果主机 USB 模块检测到一个新的 USB 设备连接时，该位置 1。该位只有当 HOSTMODEEN 为真时才有效，该中断表明一个外设现在已经连接并且必须要对其进行配置。

8. STALL 中断

在 USB 数据传输过程中，如果 USB 主机没有成功接收到数据，则 USB 设备会产生一个 STALL 中断，在 STALL 中断服务例程中，USB 设备会将相应端点挂起。

12.5.3 USB 设备(从机)编程结构

在设计 USB 设备驱动构件时，需对 USB 设备初始化及工作流程有更深入的理解。

1. USB 设备(从机)初始化

KL25 上电后会对 USB 模块进行初始化。由于 USB 模块的数据通信是由缓冲区描述符表 BDT 控制的，在初始化 USB 设备时必须在 RAM 中开辟连续的 512 字节作为 BDT(可根据实际使用的端点数修改 BDT 的大小)，而 USB 模块是通过页寄存器来指向 BDT，所以在 USB 初始化过程中(使能 USB 模块前)必须设置页寄存器，其作用是定位到端点的缓冲区描述符表。在接收到数据或发送数据时，USB 模块就能根据该寄存器及状态寄存器找到该端点的缓冲区描述符表项，并将数据存放到对应的缓冲区或将缓冲区描述符表项指定的发送缓冲区的内容发送出去。

2. USB 设备(从机)枚举

此时将 KL25 的 USB 端口与 PC 的 USB 端口相连，PC 将开始对 KL25 进行枚举。主机会根据获取的 VID 和 PID 以及其他相关信息识别一个 USB 设备，因此即使是 VID 和 PID 相同的 USB 设备也可以被 PC 区分开(正规厂商生产的 USB 设备，不会存在 VID 和 PID 都相同的两个设备)。

3. USB 设备(从机)数据收发

USB 模块根据三个页寄存器(USBx_BDTPAGE1、2、3)和状态寄存器找到对应的 BDT，并将对缓冲区进行操作，收发的长度为 BDT 中指定的数据长度(最大为 32)。

USB 设备数据发送程序只要将待发送数据放入缓冲区，然后将发送缓冲区的地址及数据长度写到指定端点的 BDT 中，修改 BC、DTS 字段即可启动数据传输。

USB 模块接收到数据后自动存放在 BDT 所指定的缓冲区中，并将接收的实际长度存放在 BDT 的 BC 域中，在接收函数中先读取 BDT 可获取收到的数据长度，然后读取缓冲区，将内容取出放入特定变量中。

4. USB 设备(从机)中断处理流程

在 USB 通信协议中,所有的通信都是由 USB 主机发起的,USB 设备不能主动地发送数据,只能在收到主机发来的令牌后进行中断响应。因此 USB 设备初始化完成后的数据收发,甚至设备枚举等均在 USB 设备中断服务例程中完成,图 12-14 详细描述了 USB 设备对中断响应的流程。

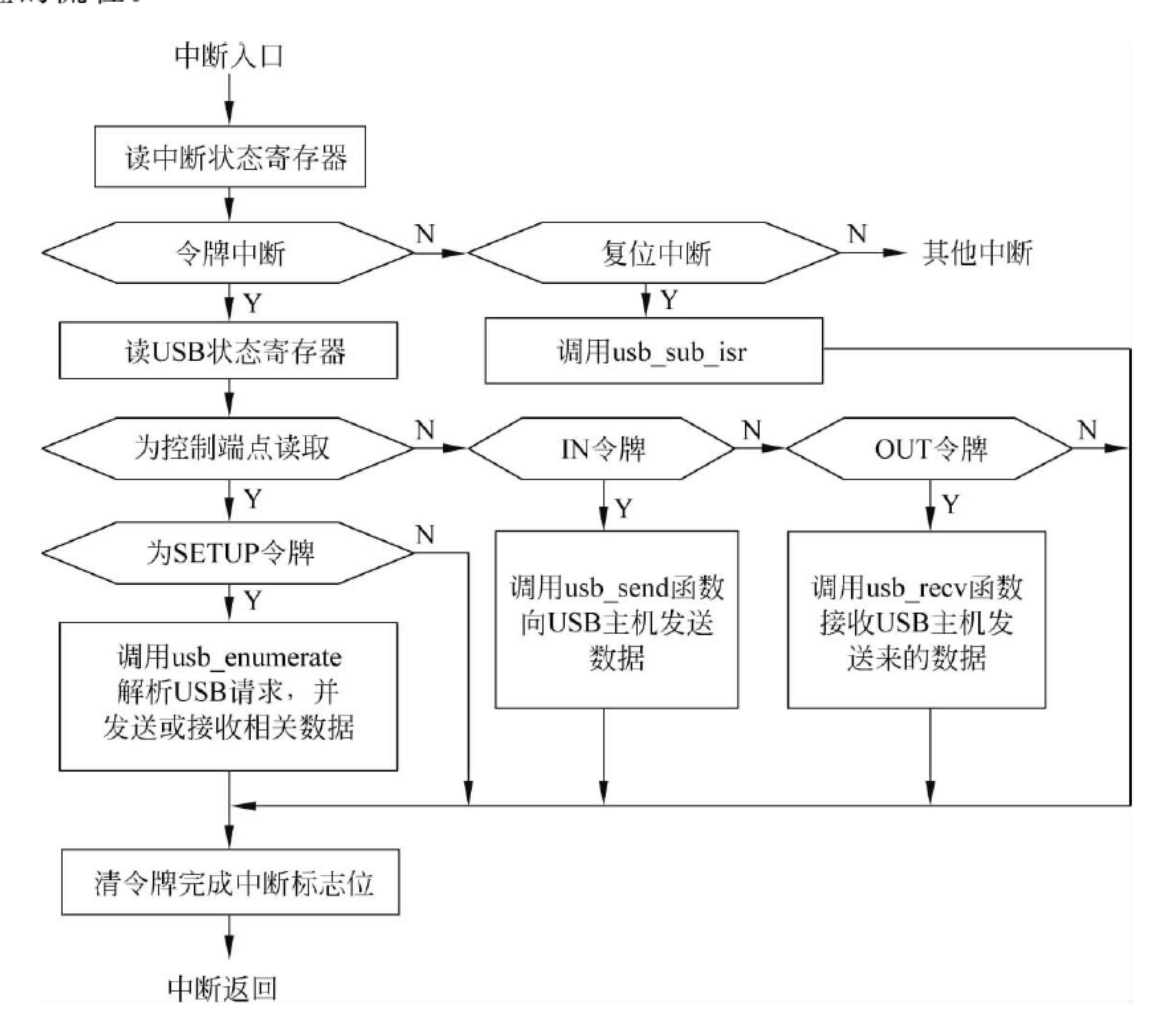


图 12-14　USB 中断服务例程流程图

12.5.4　USB 主机编程结构

1. USB 主机程序层次结构

KL25/26 的 USB 主机程序比较复杂,为了使程序更加清晰和规范,按照驱动构件化设计思想,将 USB 主机程序分为 4 个层次,并且每个层次均按照构件化设计思想将函数进行封装。这 4 层分别是文件系统层、USB 类层、USB 设备层和 USB 驱动构件层,从图 12-15 中可以看到各层之间的关系。

在制作 USB 主机构件之前必须掌握其工作流程,下面通过查询方式详细描述 USB 主机驱动构件的工作步骤。首先进行主机初始化,使能主机模块。当主机检测到有 USB 设备连接时,按照以下步骤完成对设备的初始化。

(1) 检测确实有设备连接。初次检测到有 USB 设备连接后需要按照 USB 协议清除 ATTACH 标志并等待 100ms 再次检测总线上是否有 USB 设备连接。

(2) 在确认有 USB 设备连接后需复位总线,这时 USB 设备可以使用默认地址与主机通过控制传输进行通信。

(3) 取得设备描述符。主机调用 USBGetDeviceDesc 函数发送 IN 包，从机接收到该包后调用 usb_decode_setup 函数进行解包，分析后知道是读取描述符请求的标准请求，会将设备描述符发送给主机。在设备描述符中指明了从机设备有几个配置描述符。

(4) 取得配置描述符。主机调用 USBGetConfigDesc 函数发送 IN 包，从机接收到该包后调用 usb_decode_setup 函数进行解包，分析后知道是读取配置描述符请求的标准请求，会将 usb_config_descriptor[]的配置信息发送给主机。配置信息包括配置描述符、接口描述符以及端点描述符。在本例中主机会收到 32 字节的数据，包括一个配置描述符、一个接口描述符以及两个端点描述符。

(5) 查找接口描述符。由于配置描述符中可以有多个接口描述符，需要从中选择主机支持的接口。

(6) 查找端点描述符。接口描述符中指定了所支持的端点数，需要取得分别支持 IN 和 OUT 的端点。

(7) 当上述的(4)～(6)均满足时就可以按照该配置信息对从机进行设置，如主机发送 USBSetConfig(1)，从机调用 usb_decode_setup 函数进行解包后会对其进行设置。如果不满足时需要取得另外的配置描述符再按照(4)～(6)进行查找。

(8) 完成了上面步骤后主机就知道从机的端点支持数据读取和写入的长度，以及如何从设备的端点读取和写入数据，即可以根据配置信息与从机进行通信了。当从机的接收端点对应的 BD 交给 USB 模块后，主机通过调用 USBWriteData 函数，就能向设备写入数据，否则设备不能接收到数据。同样的道理，设备的发送端点对应的 BD 交给 USB 模块后，主机通过调用 USBReadData 函数，就能读取设备从对应的缓冲区中发送的数据。

图 12-15 是 USB 主机读数据和写数据的流程。

因此作为 USB 主机的 KL25/26 的 USB 驱动程序应具有使能主机模式、与接入设备完成控制传输、向目标设备发送一个块数据的功能。

2. USB 主机初始化

(1) 使能主机模式(CTL[HOSTMODEEN]=1)。使能下拉电阻，禁止上拉电阻。开始产生 SOF 向 SOF 计数器中写入 12 000。向 USB 使能位写 0(CTL[USBENSOFEN]=0)禁止产生开始帧包。

(2) 使能 ATTACH 中断(INTEN[ATTACH]=1)。

(3) 等待 ATTACH 中断(ISTAT[ATTACH])。USB 目标设备上拉电阻改变 DPLUS 或 DMINUS 状态，从 0 变为 1(SE0 变为 J 或 K 状态)。

(4) 检测控制寄存器的 JSTATE 和 SE0 的状态。如果 JSTATE 位为 0 说明所连接的设备为低速设备。如果所连接的设备为低速设备则将地址寄存器的低速位(ADDR[LSEN])以及端点 0 控制寄存器的不带 HUB 的主机位(ENDPT0[HOSTWOHUB])置 1。

(5) 使能 RESET(CTL[RESET])10ms。

(6) 使能 SOF 包以防止接入设备从运行转到挂起状态(CTL[USBENSOFEN]=1)。

3. 与接入设备完成控制传输

(1) 将端点控制寄存器设置为双向控制传输，即 ENDPT0[4:0]=0x0d。

(2) 将设备框架建立命令复制到存储器缓冲区。设备框架命令集见 USB 2.0 协议。

(3) 初始化端点 0 的 BDT 以发送 8 字节的设备框架命令数据(如 Get_Descriptor)。首

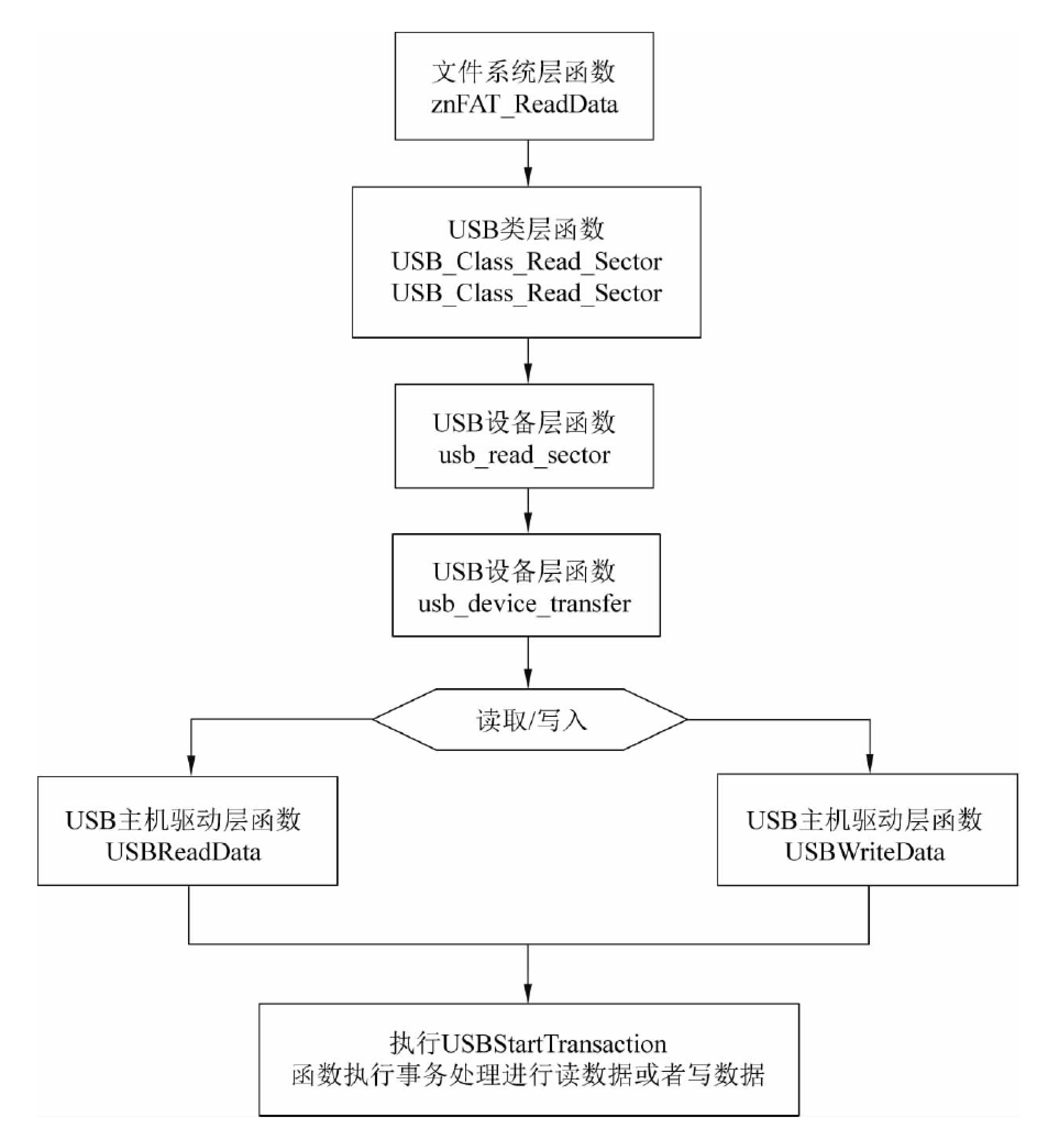


图 12-15　USB 主机读写数据流程

先将 BDT 命令字设为 0x00080080(字节数为 8,OWN 位为 1),然后将 BDT 缓冲区地址域设为该 8 字节命令缓冲区的开始地址。

(4) 设置 USB 地址为地址寄存器中的值(ADDR[6:0])。USB 总线复位后,USB 设备地址为 0。通过设置地址设备框架命令会将其设置为其他的值(通常为 1)。

(5) 向端点 0 即目标设备的默认控制管道的令牌寄存器写入 SETUP(TOKEN＝0xD0),从而在总线上发送 SETUP 令牌,并紧跟一个数据包。该数据包传输结束后,在 BDT 的 PID 域返回一个握手包。向 BDT 写入后,会产生令牌结束中断(ISTAT [TOKDNE]),结束 SETUP 事务的设置阶段。

(6) 开始 SETUP 事务的数据阶段,在存储器中为发送建立缓冲区。

(7) 初始当前端点 0 的 BDT 用于发送数据。首先,将 BDT 命令字设为 0x004000C0(字节数为 64,OWN 位为 1,DATA 触发为 DATA1),然后,将 BDT 缓冲区地址域设置为数据缓冲区的开始地址。

(8) 向端点 0 写入 IN 或 OUT 令牌,后跟数据包。数据包结束后,写 BDT 并产生一个

令牌结束中断(ISTAT[TOKDNE])。对于单个包数据阶段,这就结束了SETUP事务的数据阶段。

(9) 开始SETUP事务的状态阶段,在存储器中建立一个用于接收或发送0长度数据包的缓冲区。

(10) 初始化端点0的BDT用于发送状态数据。首先,将BDT命令字设为0x00000080(字节数为0,OWN位为1,DATA触发为Data0),然后,将BDT缓冲区地址域设置为数据缓冲区的开始地址。

(11) 向端点0写入IN或OUT令牌,后跟一个0长度数据包。数据包结束后,会将设备发出的握手包写入BDT并产生一个令牌结束中断(ISTAT[TOKDNE])。这就结束了SETUP事务的状态阶段。

4. 向目标设备发送一个全速块数据

(1) 完成发现接入设备的所有步骤并配置接入的设备。向ADDR寄存器写入目标设备的地址。典型情况是USB总线上只有一个设备,所以ADDR值为0x01并保持不变。

(2) 设置ENDPT0寄存器为0x1D,使能带握手的发送和接收。

(3) 设置缓冲区描述符表中端点0的偶数发送项为最多发送64字节。

(4) 设置目标USB设备地址为地址寄存器的值(ADDR[6:0])。

(5) 将TOKEN寄存器设为OUT。这将触发OTG模块发送令牌和数据。

(6) 设置缓冲区描述符表中端点0的奇数发送项为最多发送64字节。

(7) 跟第(4)步一样,向TOKEN寄存器写入OUT令牌。

(8) 等待令牌完成中断。

(9) 产生令牌完成中断后,可以检查BDT并且返回到第(2)步。

12.6 KL25/26芯片作为USB设备(从机)的驱动构件设计

USB设备驱动构件源文件(usb.c)如下。

```
#include "usb.h"

//========================================================
//函数名:usb_init
//功能: USB模块初始
//参数:Device_Name: USB设备名
//返回: 无
//========================================================
void usb_init(uint_8 Device_Name[])
{
//后面加0x200UL,防止占用rom区
BDTtable = (tBDT *)((( uint_32 ) BDT & 0xFFFFFE00UL)+0x200UL);
//USB相关字符串初始化
USB_String_Table.String_Descriptor0=String_Descriptor0;
USB_String_Table.String_Descriptor1=String_Descriptor1;
```

```
USB_String_Table.String_Descriptor2=String_Descriptor2;
USB_String_Table.Device_Name=Device_Name;
//将端点 0 接收奇缓冲区的地址存储到 setup 数据包的指针对象 Setup_Pkt 中
//便于端点 0 将接收到的 PC 发来的 setup 包
//指向端点 0 的接收奇缓冲区首地址
    Setup_Pkt=(tUSB_Setup *)BD_BufferPointer[bEP0OUT_ODD];
    gu8USB_State=uPOWER;                                  //USB 设备处于上电状态
    //USB_FMC_ACC_ENABLE;
    USB_REG_SET_ENABLE;
    USB_REG_CLEAR_STDBY;
    //MPU_CESR=0;禁止 MPU
    FLAG_SET(SIM_SOPT2_PLLFLLSEL_SHIFT,SIM_SOPT2);    //使能 PLL 输出
    FLAG_SET(SIM_SOPT2_USBSRC_SHIFT,SIM_SOPT2);       //使能 PLL/FLL 为时钟源
    //SIM_CLKDIV2|=USB_FARCTIONAL_VALUE;              //USB 分频因子设置
    SIM_SCGC4|=(SIM_SCGC4_USBOTG_MASK);               //USB 模块时钟门使能
    enable_irq(USB_INTERRUPT_IRQ);                    //使能 USB 模块 IRQ 中断
    //USB 模块寄存器配置、USB0_USBTRC0:收发控制寄存器
    USB0_USBTRC0 |= USB_USBTRC0_USBRESET_MASK;
    //等待 USB 模块复位发生,发生就退出循环
    while(FLAG_CHK(USB_USBTRC0_USBRESET_SHIFT,USB0_USBTRC0)){};
    //配置 BDTPAGE1,2,3 寄存器,设置 BDT 基址寄存器
    //(低 9 位是默认 512 字节的偏移) 512 = 16 * 4 * 8
    //8 位表示:4 个字节的控制状态,4 个字节的缓冲区地址
    USB0_BDTPAGE1=(uint_8)((uint_32)BDTtable>>8);
    USB0_BDTPAGE2=(uint_8)((uint_32)BDTtable>>16);
    USB0_BDTPAGE3=(uint_8)((uint_32)BDTtable>>24);
    //清除 USB 模块复位标志(之前发生复位会使该位置 1)
    //检测到 USB 复位,置 1,通知 MPU 向地址寄存器写入 0x00,
    //并使能端点 0
    FLAG_SET(USB_ISTAT_USBRST_MASK,USB0_ISTAT);
    //使能 USB 模块复位中断
    FLAG_SET(USB_INTEN_USBRSTEN_SHIFT,USB0_INTEN);
    USB0_USBCTRL = 0x40;                        //USB 的 SE0 信号,两根数据线被拉低
    USB0_USBTRC0 |= 0x40;               //必须:强制设置第 6 位为 1,USB 收发控制寄存器
    //上拉使能,这样主机才能识别到设备并对其进行配置
    FLAG_SET(USB_CONTROL_DPPULLUPNONOTG_SHIFT,USB0_CONTROL);
    USB0_CTL |= 0x01;                   //USB 模块使能位
}
//==========================================
//函数名:usb_enumerate
//功能:USB 枚举,用于处理 USB 设备复位后 USB 主机发送来的设备请求
//参数:无
//返回:无
//==========================================
void usb_enumerate()
{
uint_8 u8IN;
u8IN=USB0_STAT & 0x08;              //u8IN 表示端点 0,最后更改的 BD 位于 BDT 的偶数行
//上行传输
if(u8IN)    usb_ep0_in_handler();
```

```
//下行传输
else
{
//接收到 0x0D,表示是 SETUP 包
if(BDTtable[bEP0OUT_ODD].Stat.RecPid.PID == mSETUP_TOKEN)
usb_setup_handler();
else
            usb_ep0_out_handler();
}
}

//==========================================================
//函数名: usb_send
//功能: USB 发送数据
//参数: SendBuff: 待发数据缓冲区
//        DataLenght: 待发数据长度
//返回: 无
//备注: 一次性传输的数据长度是端点所支持的最大数据长度(32 字节),如果发送的数据长度大于
//32 字节,则分为多次传输。
//==========================================================
uint_8 usb_send(uint_8 * SendBuff, uint_8 * DataLength)
{
    uint_8 i, counter;
    uint_8 * pBuffer;
    uint_32 vEP2Idx = 0;
    pBuffer = gu8EP2_IN_ODD_Buffer;
    //判断发送的数据长度是否大于端点所支持的最大数据长度
if( * DataLength > EP2_SIZE)
counter = EP2_SIZE;
else
counter = * DataLength;
for(i=0; i<counter; i++, vEP2Idx++)
    //将待发送数据复制到对应端点发送缓冲区
pBuffer[i] = SendBuff[vEP2Idx];
BDTtable[bEP2IN_ODD].Cnt = counter;
//异或同 0 异 1   vEP2State = kUDATA0   0x88
vEP2State ^= 0x40;
//表示 USB 拥有操控权,接收了 DATA0 包,当前作为 ACK 握手包
BDTtable[bEP2IN_ODD].Stat._byte= vEP2State;
//将未处理的数据存放到待发缓冲区
 * DataLength = * DataLength-counter;
for(i = 0;i <( * DataLength);i++)
{
SendBuff[i] = SendBuff[i + counter];
}
if(counter)
return counter;
else
return 0;
}
```

```
//=========================================
//函数名: usb_recv
//功能: USB 接收数据
//参数: Recvbuff:接收数据缓冲区
//        DataLength:接收的数据长度
//返回: 成功: 返回接收数据的长度; 失败: 返回 0
//=========================================
uint_8 usb_recv(uint_8 * RecvBuff,uint_8 * DataLength)
{
uint_8 i;
    uint_8 * pBuffer;
     * DataLength=0;
    pBuffer = gu8EP3_OUT_ODD_Buffer;
    //接收到数据后,BD 的 BC 字段是接收到数据的长度
    if(BDTtable[bEP3OUT_ODD].Cnt != 0)
    {
for(i = 0;i < BDTtable[bEP3OUT_ODD].Cnt;i++)
{
//将对应端点接收缓冲区数据复制到接收数据区
RecvBuff[(* DataLength)] = pBuffer[i];
(* DataLength)+=1;
}
    }
    //异或同 0 异 1   vEP2State = kUDATA1   0xC8
    vEP3State ^=0x40;
    //表示 USB 拥有操控权,接收了 DATA1 包,作为 ACK 握手包
    BDTtable[bEP3OUT_ODD].Stat._byte=vEP3State;
    BDTtable[bEP3OUT_ODD].Cnt = EP3_SIZE;
    if(* DataLength>0)   //接收数据成功
return (uint_8) * DataLength;
else
return 0;
}
```

12.7　KL25/26 芯片作为 USB 主机的驱动构件设计

USB 主机驱动构件源文件(usb_host.c)如下。

```
#include "usb_host.h"
//=========================================
//函数名:USBHostInit
//功能: USB 模块初始化,配置为 USB 主机模式
//参数: 无
//返回: 0=成功; 非 0=异常
//=========================================
uint_8 USBHostInit(void)
```

```
{
    uint_8 ep;
    uint_16 i;

    //指向 BDT,避免占用 rom 区
    tBDTtable = (tBDT *)((( uint_32 ) tBDT_unaligned & 0xFFFFFE00UL) + 0x200UL);
    bdt = (uint_8 *)tBDTtable;
    //设备相关信息
    my_device.address = INVALID_ADDRESS;                //无设备时是一个无效地址
    ep_info.next_tx = 0;
    ep_info.next_rx = 0;
    for(i = 0;i < 512;i++)
    {
        bdt[i] = 0;                                     //将 512 个 DB 清零
    }
    //端点相关信息
    for(ep=0;ep<MAX_EP_PER_DEVICE;ep++)
    {
        my_device.eps[ep].address = INVALID_ADDRESS;    //端点地址
        my_device.eps[ep].last_due = 0;                 //最近到期
        my_device.eps[ep].interval = 74;                //间隔
    }
    SIM_SOPT2 |= SIM_SOPT2_USBSRC_MASK | SIM_SOPT2_PLLFLLSEL_MASK;
    SIM_SCGC4 |= SIM_SCGC4_USBOTG_MASK;
    enable_irq(USB_INTERRUPT_IRQ);
    USB0_USBTRC0 |= USB_USBTRC0_USBRESET_MASK;
    while (USB0_USBTRC0 & USB_USBTRC0_USBRESET_MASK){}
    USB0_USBCTRL = 0x40;                                //若 D+,D-上的弱下拉使能
    USB0_ISTAT = 0xFF;
    USB0_CTL |= USB_CTL_ODDRST_MASK;                    //清 odd,指向 even 区
    USB0_BDTPAGE1 = (uint_8)(((uint_32)BDT_BASE) >> 8);
    USB0_BDTPAGE2 = (uint_8)(((uint_32)BDT_BASE) >> 16);
    USB0_BDTPAGE3 = (uint_8)(((uint_32)BDT_BASE) >> 24);
    USB0_SOFTHLD = USB_SOFTHLD_CNT_MASK;                //SOF 计数器阈值
    USB0_OTGCTL = USB_OTGCTL_DPLOW_MASK | USB_OTGCTL_DMLOW_MASK |
USB_OTGCTL_OTGEN_MASK; //上下拉 OTG 寄存器使能
    USB0_USBCTRL |= USB_USBCTRL_PDE_MASK;               //弱下拉使能,禁止上拉电阻
    USB0_USBCTRL &= ~USB_USBCTRL_SUSP_MASK;             //清挂起状态
USB0_INTEN = USB_INTEN_ATTACHEN_MASK;                   //连接中断使能
    //USB0_CTL |= USB_CTL_USBENSOFEN_MASK;              //禁止产生开始帧包
    USB0_USBTRC0|=0x40;
    USB0_CTL = USB_CTL_HOSTMODEEN_MASK;                 //使能主机模式
    return 0;
}

//==========================================================
//函数名:InitUSBDevice
//功能: 初始化接入的 USB 设备
//参数: 无
//返回: 0=成功; 1=异常
```

```
//备注: 使用回调函数 usb_device_transfer,用于传输数据
//==========================================
uint_8 InitUSBDevice(uint_8 * device_inf)
{
    uint_8 lun = 0;
    for(;;)                                     //等待发生 ATTACH 中断,设备连接检测到
    {
        if((USBFlag & USB_ISTAT_ATTACH_MASK) != 0)
        {
            USBFlag &= ~USB_ISTAT_ATTACH_MASK;
            break;
        }
    }
    USBFlag = 0;
    lun=(uint_8)usb_mst_init();
    if (lun)
    {
        scsi_open_device();
        scsi_get_device_string(device_inf, 32);
    }
    else
    {

        return 0;
    }
    return 1;
}

//==========================================
//函数名: USBReadData
//功能:  USB 数据读取
//参数:  ep:USB 端点号
//         length:读的数据长度
//         ReadBuffer:存放读数据的缓冲区
//返回: 0=成功; 1=失败
////备注: 调用内部函数 USBStartTransaction 执行数据的读取,读取的数据可以在多个事
//         务处理中,因此需要根据端点所支持的最大数据包的长度决定执行多少次事务处理
//==========================================
uint_16 USBReadData(uint_8 ep, uint_16 length,uint_8 *  ReadBuffer)
{
    uint_32 curr=0;
    uint_16 got,psize;
    MKDBG_TRACE(ev_receive, ep);
    if ((USB0_CTL & USB_CTL_HOSTMODEEN_MASK) == 0)
    {
        tr_error=tre_not_running;
        return(0);
    }
    while(curr < length)
    {
```

```
            psize=(uint_16)(MIN(my_device.eps[ep].psize, length));
            //开始执行 USB 事务处理函数,进行数据的读取
            got   = USBStartTransaction(TRT_IN, ReadBuffer+curr, psize, ep);
            if (got == ((uint_16)-1u))
            {
                break;
            }
            curr += got;
            if (got != my_device.eps[ep].psize)
            {
                break;
            }
        }
        return(curr);
}

//===========================================================================
//函数名: USBWriteData
//功能:   向 USB 设备(U 盘)写入数据
//参数: ep:USB 端点号
//        length:要写入的数据长度
//        buff:存放要写入的数据的缓冲区
//返回: 0=成功; 1=失败
//备注: 调用内部函数 USBStartTransaction 执行数据的写入,写入的数据可以在多个事
//        务处理中,因此需要根据端点所支持的最大数据包的长度决定执行多少次事务处理
//===========================================================================
uint_8 USBWriteData(uint_8 ep,uint_16 length,uint_8 * WriteBuffer)
{
        uint_32 curr=0;
        uint_16 psize,r;
        MKDBG_TRACE(ev_send, ep);
        if ((USB0_CTL & USB_CTL_HOSTMODEEN_MASK) == 0)
        {
            tr_error=tre_not_running;
            return(0);
        }
        if(length > 512)
            return 0;
        while(curr<length)
        {
            psize = (uint_16)(MIN(my_device.eps[ep].psize, length));
            //开始执行 USB 事务处理函数,进行数据的写入
            r = USBStartTransaction(TRT_OUT, WriteBuffer+curr, psize, ep);
            if (r!=psize)
            {
                CMX_ASSERT(r==((uint_16)-1u));
                break;
            }
            curr += psize;
        }
```

```
    return(curr);
}

//=========================================
//函数名: CheckUSBDeviceStatus
//功能:  检测 USB 设备状态
//参数: 详见 SetupPack 包的包结构
//返回: 0=成功; 1=失败
//=========================================
uint_8 CheckUSBDeviceStatus()
{
    int i = 0;
    USB0_ISTAT = USB_ISTAT_ATTACH_MASK;
    for(i = 0;i < 1000;i++)
    {
    }
    if(!(USB0_ISTAT & USB_ISTAT_ATTACH_MASK))
    {
        return 0;
    }
    else
    {
        return 1;
    }
}
```

第 13 章　系统时钟与其他功能模块

本章导读：本章主要介绍了基本功能模块外的其他功能模块，主要内容有：①系统时钟的概述与设置；②电源模块；③低漏唤醒单元；④位带操作；⑤看门狗模块；⑥复位与启动模块。这些内容一般会在程序的初始化中使用。与前面的章节相比，本章内容较杂且难理解，但对于 KL25 芯片的开发，又是必不可少、需要全面理解的部分。

本章参考资料：13.1 节(系统时钟的概述与设置)总结自《KL25 参考手册》的第 5 章；13.2 节(电源模块)总结自《KL25 参考手册》的第 13 章；13.3 节(低漏唤醒单元)总结自《KL25 参考手册》的第 15 章；的第 17 章；13.4 节(看门狗模块)总结自《KL25 参考手册》的第 3 章、第 5 章及第 12 章等；13.5 节(复位与启动模块)总结自《KL25 参考手册》的第 6 章；13.6 节(位带操作)总结自《KL25 参考手册》的第 17 章。

13.1　时 钟 系 统

时钟系统是微控制器(MCU)的一个重要部分，它产生的时钟信号要贯穿整个芯片。时钟系统设计得好坏关系到芯片能否正常工作。KL25 芯片提供多个时钟源选择，每个模块可以根据自己的需求选择对应时钟源。

13.1.1　时钟系统概述

KL25 芯片的时钟系统由振荡器(Oscillator，OSC)、实时时钟(Real Time Clock，RTC)、多功能时钟发生器(Multipurpose Clock Generator，MCG)、系统集成模块(System Integration Module，SIM)和电源管理器(Power Management Controller，PMC)等模块组成。其中，OSC 和 RTC 模块通过外接的晶振器件为系统引入外部参考时钟信号，MCG 模块为系统中的各模块分配时钟源，SIM 模块为系统中的各模块选择时钟源，PMC 模块可输出 1kHz 的参考时钟信号。时钟系统的框图如图 13-1 所示。

OSC 提供了一组外部接口(XTAL 与 EXTAL)，RTC 模块提供了一个外部引脚 RTC_CLKIN，两者都用于接入外部参考时钟信号。OSC 可使用晶体振荡器，也可使用其他时钟信号源。OSC 模块系统主要使用外部时钟源(例如 50MHz 有源振荡器)。而 RTC_CLKIN 一般只能接 32.768kHz 的外部时钟信号。

多功能时钟发生器 MCG 模块为 MCU 提供了多种时钟源选择，内部包含一个锁频环(Frequency-Locked Loop，FLL)和一个锁相环(Phase-Locked Loop，PLL)，分别对内部参考时钟信号和外部参考时钟信号进行倍频。其中，FLL 接收内部或外部的参考时钟源，MCG 模块可提供 4MHz 和 32kHz 两种内部参考时钟信号；PLL 接收外部参考时钟源，外部参考时钟来自 OSC 模块。MCG 模块可以选择 FLL 或 PLL 的输出时钟 MCGFLLCLK 或 MCGPLLCLK，或者内部参考时钟的输出时钟 MCGIRCLK。

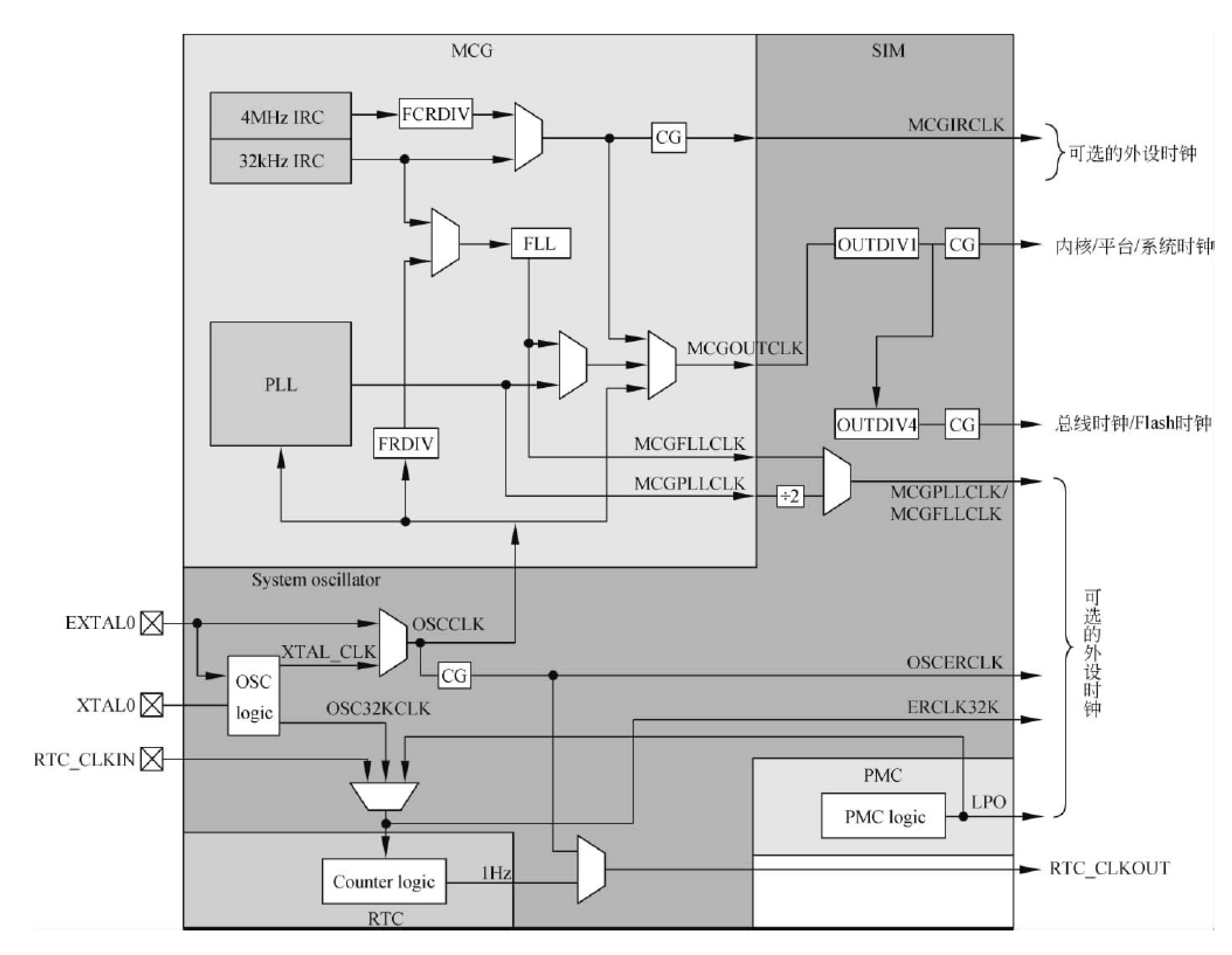


图 13-1　时钟系统框图

MCG 模块管理的系统时钟源可作为一些特定模块的时钟驱动信号。时钟产生模块将选定的时钟源分频，产生多种不同的时钟频率供多个模块使用，例如，内核/平台/系统时钟、总线时钟和 Flash 存储器时钟等。时钟产生模块也实现了模块专用的时钟门控逻辑，允许通过停止供应时钟信号关闭模块的功能。

某些模块拥有本身专用的时钟源，例如 USB OTG 控制器，这些时钟可取自于 MCG 锁相环时钟 MCGPLLCLK 或 MCG 锁频环时钟 MCGFLLCLK。另外，还有一些模块可以选择不同的专用时钟源，例如实时时钟（Real Time Clock，RTC）模块，就使用专用的 RTC OSC 时钟源。对于大多数模块的时钟源选择是由 SIM 模块的系统选项寄存器 SOPT 中设定。

13.1.2　时钟模块概要与编程要点

时钟源的选择和复用是通过 MCG 模块来控制和编程的，而系统的时钟分频器和模块时钟门是通过 SIM 模块来编程设置的。

内部参考时钟 MCGIRCLK 是由 4MHz 的高速内部参考时钟经过分频（由 MCG 状态控制寄存器 MCG_SC[FCRDIV]设定分频因子），或者 32kHz 的低速内部参考时钟提供，低功耗模式下使用内部参考时钟。两者经过内部参考时钟选择位（MCG_C2[IRCS]）选择后，经过打开的时钟门内部参考时钟使能位（MCG_C1[IRCLKEN]）给外设提供时钟源。

外部参考时钟（OSCERCLK）可以由外部晶振提供时钟源，通过设置外部参考使能

(OSC_CR[ERCLKEN])位可以打开它的时钟门。通过置外部参考时钟选择位(MCG 控制寄存器 MCG_C2[EREFS0])选择外部晶振作为时钟源。

ERCLK32K 可以由 RTC_CLKIN、OSC32KCLK 以及 1kHz LPO 提供时钟源。通过选择 32K 晶振选择位(SIM_SOPT1[OSC32KSEL])可以为 ERCLK32K 选择时钟源。OSC32KCLK 可以由外部晶振获得。

RTC_CLKOUT 可以选择 RTC 1Hz 和 OSCERCLK 驱动。通过 SIM_SOPT2 中 RTC 输出选择位(RTCCLKOUTSEL)选择 RTC 1Hz 还是 OSCERCLK。RTC 1Hz 的时钟源和 ERCLK32K 一致。LPO 只能用于计时唤醒功能。只有当外部晶振是 32kHz 且晶振低功耗模式选择时,RTC_CLKOUT 才能选择 OSCERCLK 作为时钟源。除了 VLLS0 模式,OSCERCLK 可以在任何模式下使用。在 VLLS0 模式下,只能使用 RTC_CLKIN。

晶振模块的输出 OSCCLK 一般经过分频后进入 FLL(锁频环)或 PLL(锁相环)进行倍频处理,经过 PLL 得到 MCGPLLCLK,经过 FLL 得到 MCGFLLOUT。MCGOUTCLK 可由内部参考时钟,PLL/FLL 或者外部参考时钟也就晶振时钟(OSCCLK)经过时钟选择位(MCG_C1[CLKS])选择作为时钟源。经过分频(OUTDIV1)和一个时钟门输出作为系统时钟(内核时钟/平台时钟)。再经过 SIM_CLKDIV1[OUTDIV4]和一个时钟门输出作为总线时钟和 Flash 时钟。MCGFLLOUT 和 MCGPLLOUT/2 经过设置 SIM 中的 PLLS 选择位来作为外设时钟源。FLL 相比于 PLL 不是很精确,所以在 MCU 没有很严格的时钟要求时采用 FLL。而在需要有精确的时钟要求时最好采用 PLL。

多用途时钟信号生成器(MCG)模块为 MCU 提供多种时钟源选择。这个模块由一个锁频环(FLL)和一个锁相环(PLL)组成。FLL 可由一个内部或外部参考时钟控制,而 PLL 可由一个外部参考时钟控制。这个模块可以选择 FLL 或 PLL 输出时钟,或者内部或者外部参考时钟作为 MCU 系统时钟源。

MCG 共有 9 种运行模式:FEI,FEE,FBI,FBE,PBE,PEE,BLPI,BLPE 和 STOP。需要注意的是,这 9 种模式不是可以任意切换的,比如不可以直接从 FEI 切换到 PEE,模式的切换需要遵守图 13-2,只有图 13-2 中给出的切换才是允许的。9 个状态在图 13-2 中表示。这些箭头表明允许的 MCG 模式转换。表 13-1 分别给出各种模式的含义。

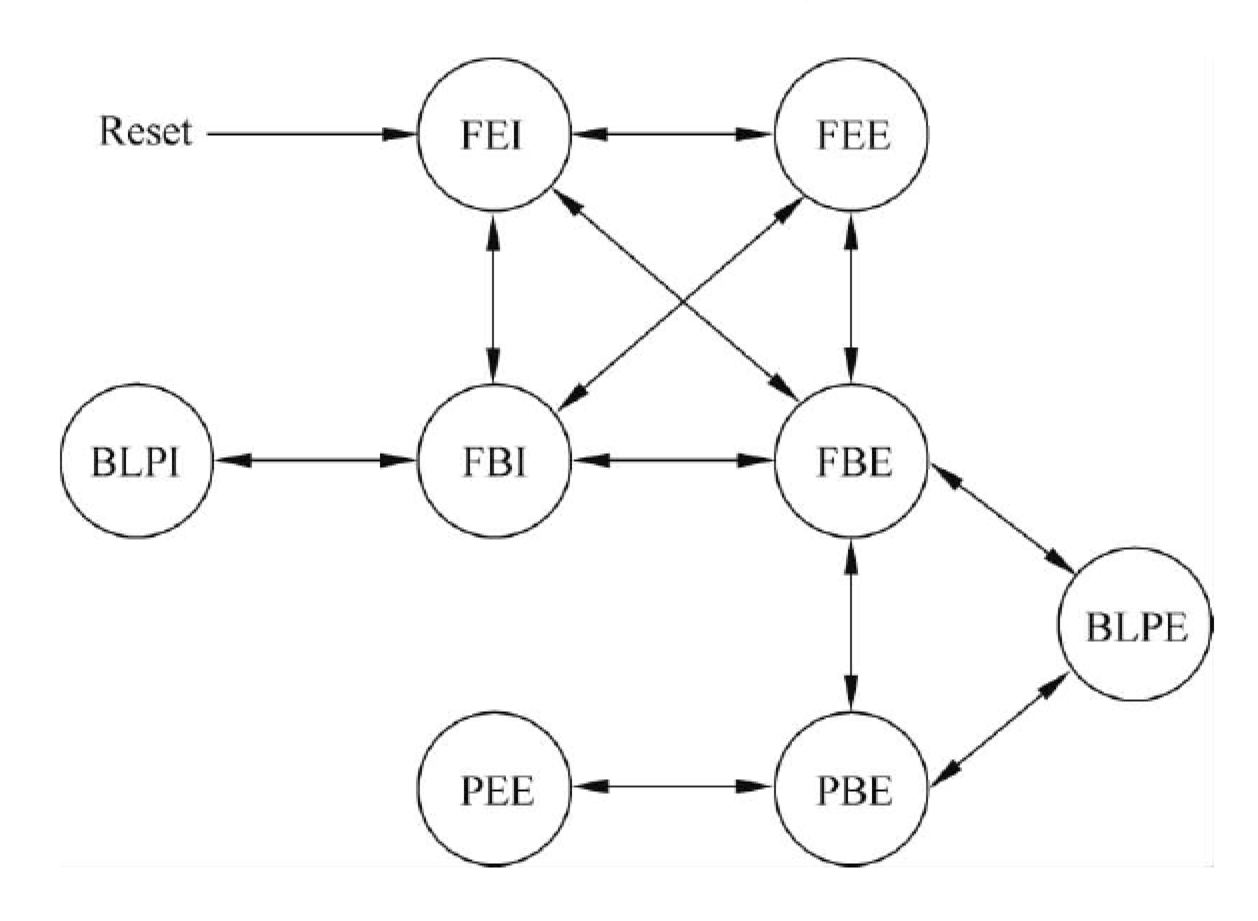


图 13-2　MCG 模式状态图

表 13-1　MCG 各种模式的含义

模　式	说　明
FEI	FLL 内部忙碌。FLL 使能，并且采用内部参考时钟的模式。此种模式下，MCG 内部 FLL 使能，它使用 32kHz 内部参考时钟作为时钟源，产生系统运行所需要的各种时钟。FEI 是上电后的默认模式
FEE	FLL 外部忙碌。FLL 使能，并且采用外部参考时钟的模式。此种模式下，MCG 内部 FLL 使能，它使用外接的晶振作为时钟源
FBI	FLL 内部旁路，其输出没有被使用；MCGOUT 从低速(32kHz)或者高速(4MHz)内部参考时钟获得
FBE	FLL 外部旁路，其输出没有被使用，直接使用外部参考时钟作为 MCGOUTCLK 给系统提供时钟
PEE	PLL 外部忙碌。PLL 使能，且使用外部参考时钟作为 PLL 时钟源的模式。此种模式下，系统运行所需时钟是由 PLL 提供的
PBE	PLL 外部旁路，其输出没有被使用；而是直接用外部参考时钟给系统运行提供时钟
BLPI	内部旁路低功耗。这是一种低功耗模式；此时 PLL 和 FLL 均不工作，系统运行所需时钟由内部参考时钟提供
BLPE	外部旁路低功耗。与 BLPI 相似，唯一区别是，系统运行时钟由外部参考时钟提供
STOP	STOP 模式。STOP 模式是一种特殊模式，只要 MCU 进入 STOP 模式，那么 MCG 也就进入 STOP 模式，当 MCU 从 STOP 模式退出，除非在 STOP 模式时发生复位，否则 MCG 也将退出 STOP 模式，重新进入先前的模式(即 MCG 进入 STOP 模式前的模式)

本节测试实例选择从 FEI 进入 FBE，再进入 PBE，最后达到 PEE 状态。具体步骤如下。

(1) 首先，FEI 必须过渡到 FBE 模式。

C2＝0x1C，因为我们的核心板上采用的外部晶振为 8MHz，属于高频率范围，C2[RANGE]设置为 0b01。因为正在使用的外部参考时钟源是晶振，所以 C2[EREFS]设置为 1。

C1＝0x98，C1[CLKS]设置为 0b10，以便选择作为系统时钟源的外部参考时钟。C1[FRDIV]设置为 0b011，对应 256 分频因为 8MHz/256＝31.25kHz，在由 FLL 要求的 31.25～39.0625kHz 频率范围内。C1[IREFS]清除为 0，选择外部参考时钟和外部晶振。

循环直到 S[OSCINIT]为 1，表明由 C2[EREFS]选择的晶振已经被初始化。循环直到 S[IREFST]为 0，表明外部参考是当前参考时钟源。循环直到 S[CLKST]为 0b10，表明选择外部参考时钟提供给 MCG。

(2) 此时 MCG 处于 FBE 状态，然后配置 C5[PRDIV0]产生正确的 PLL 参考频率。C5＝0x01，C5[PRDIV]设置为 0b00001，对应 2 分频导致 PLL 参考频率为 8MHz/2＝4MHz。然后，FBE 必须直接转换为 PBE 模式或者先经过 BLPE 模式再转换为 PBE 模式。本例采用直接转换为 PBE 状态。

C6＝0x40，C6[PLLS]设置为 1，表明选择 PLL 输出作为 MCG 时钟源。这时 C6[VDIV]设置为 0b00000，对应倍频因子为 24，因为 4MHz×24＝96MHz，在 BLPE 模式，因为 PLL 被关闭，VDIV 位配置无关紧要。只有在 PBE 模式设置了 PLL 倍频因子的值才能改变它们。

循环直到 S[PLLST]被设置,表明 PLL 是当前 PLLS 时钟源。

循环直到 S[LOCK]被设置,表明 PLL 要求锁存。此时处于 PBE 状态。

(3) 最后,PBE 模式转换成 PEE 模式:C1=0x10,C1[CLKS]设置为 0b00 以选择作为系统时钟源的 PLL 输出。循环直到 S[CLKST]设置为 2'b11,表明在当前时钟模式 PLL 输出被选择为提供 MCGOUT。

现在,C5[PRDIV0]设置为 0b00001 表明二分频,C6[VDIV0]=0b00000 表明乘以 24,MCGOUT=[(8MHz/2)×24]=96MHz。

13.1.3 时钟模块测试实例

```
//========================================================================
//函数名称: sys_init
//函数返回: 无
//参数说明: 无
//功能概要: (1)KL25 的 EXTAL(40)、XTAL(41)接 8MHz 外部晶振,产生 PLL/FLL 输出时钟频
//                率 48MHz,内核时钟 48MHz,总线时钟 24MHz,内部参考时钟 4MHz。
//          (2)对于这些频率,sys_init.h 有相应的宏常量定义可以供编程时使用
//========================================================================
void sys_init(void)
{
    uint_32 i = 0;
    uint_8 temp_reg = 0;

 //1.首先从 FEI 模式过渡到 FBE 模式

    //C2= 0x1C,因为我们的核心板上采用了外部晶振为 8MHz,属于高频率范围,
    //C2[RANGE]设置为 0b01; C2[HGO]设为 1 以配置晶振来进行高增益操作; 因为
    //正在使用的外部参考时钟源是晶振,所以 C2[EREFS]设置为 1。
    MCG_C2 = (MCG_C2_RANGE0(1) | MCG_C2_EREFS0_MASK);
    //C1 = 0x90 ,C1[CLKS]设置为 2'b10,以便选择作为系统时钟源的外部参考时钟。
    //C1[FRDIV]设置为 3'b011,对应 256 分频,因为 8MHz/256=31.25kHz,在 FLL 要求的
    //31.25~39.0625 kHz 频率范围内。C1[IREFS]清除为 0,选择外部参考时钟和
    //外部晶振。
    MCG_C1 = (MCG_C1_CLKS(2) | MCG_C1_FRDIV(3));
    //需要等到 S[OSCINIT]被置位外部晶振才能使用
    for (i = 0 ; i < 20000 ; i++)
    {
        //如果 S[OSCINIT]在循环结束之前被置位就跳出循环
        if (MCG_S & MCG_S_OSCINIT0_MASK) break;
    }
    //等待参考时钟状态位清 0
    for (i = 0 ; i < 2000 ; i++)
    {
        //如果 IREFST 在循环结束之前被清 0 就跳出循环
        if (!(MCG_S & MCG_S_IREFST_MASK)) break;
    }
```

```
    //等待时钟状态位以显示时钟源为外部参考时钟
    for (i = 0 ; i < 2000 ; i++)
    {
        //如果 CLKST 显示外部时钟被选择,在循环结束之前就跳出循环
        if (((MCG_S & MCG_S_CLKST_MASK) >> MCG_S_CLKST_SHIFT) == 0x2)
break;
    }
 //2.现在处于 FBE 状态,使能时钟监视器,由 FBE 直接转换为 PBE 模式
    MCG_C6 |= MCG_C6_CME0_MASK;
    //配置 PLL 为 2 分频
    MCG_C5 |= MCG_C5_PRDIV0(1);
    //配置 MCG_C6 以设置 PLL 倍频因子并且使能 PLL,PLLS 位被置位来使能 PLL,MCGOUT
    //时钟源仍然是外部参考时钟
    temp_reg = MCG_C6;                    //存储当前 C6 的值(因为 CME0 位之前被置位了)
    temp_reg &= ~MCG_C6_VDIV0_MASK;//将 VDIV 清 0
    temp_reg |= MCG_C6_PLLS_MASK | MCG_C6_VDIV0(0);//重新写值到 VDIV 并且使
                                                   //能 PLL
    MCG_C6 = temp_reg;                    //更新 MCG_C6 的值
    //等待 PLLST 状态位被置
    for (i = 0 ; i < 2000 ; i++)
    {
        //如果 PLLST 在循环结束之前被置位就跳出循环
        if (MCG_S & MCG_S_PLLST_MASK) break;
    }
    //等待 LOCK 被置位
    for (i = 0 ; i < 4000 ; i++)
    {
        //如果 LOCK 在循环结束之前被置位就跳出循环
        if (MCG_S & MCG_S_LOCK0_MASK) break;
    }
  //3.现在处于 PBE 模式。最后,PBE 模式转换成 PEE 模式
    //设置核心时钟分频器 2 分频
    //设置总线时钟分频器 2 分频 (总线时钟的时钟源是核心时钟)
    SIM_CLKDIV1 = (SIM_CLKDIV1_OUTDIV1(1) | SIM_CLKDIV1_OUTDIV4(1) );
    //清 CLKS 来打开 CLKS 多路复用器来选择 PLL 作为 MCGCLKOUT
    MCG_C1 &= ~MCG_C1_CLKS_MASK;
    //等待时钟状态位更新
    for (i = 0 ; i < 2000 ; i++)
    {
        //如果 CLKST 在循环结束之前等于 3 就跳出循环
        if (((MCG_S & MCG_S_CLKST_MASK) >> MCG_S_CLKST_SHIFT) == 0x3)
break;
    }
  //4.现在处于 PEE 模式
}
```

13.2 电源模块

13.2.1 电源模式控制

系统模式控制器(SMC)提供多种可选电源模式,用户可以根据不同的功能需求来选择不同的模式。根据用户应用的功耗需求,提供了多种功耗模式,用户可以根据需要选择保留逻辑单元和存储单元的上电状态;或关闭某些逻辑单元和存储单元电源;或关闭所有逻辑单元和存储单元电源。I/O状态在所有模式操作中都会保留。表13-2描述了可使用的电源模式。

表13-2　电源模式

模　　式	描　　述
RUN	该MCU可以运行在全速模式,内部电源完全监管,在运行模式规则下,该模式被认为是正常运行模式
WAIT	关闭内核时钟。系统时钟持续工作。总线时钟,如果开启,则持续工作。保持运行监管
STOP	关闭内核时钟。系统时钟服务其他模块。来自支持外设的停止应答信号有效后,关闭总线时钟
VLPR	该模式下内核、系统、总线、Flash时钟的最大频率受限制
VLPW	关闭内核时钟。系统、总线、Flash时钟持续工作,它们的最大频率受限制
VLPS	关闭内核时钟。系统时钟服务其他模块。来自支持外设的停止应答信号后,关闭总线时钟
LLS	关闭内核时钟。系统时钟服务其他模块。来自支持外设的停止应答信号后,关闭总线时钟。通过降低内部逻辑的电压,将MCU置于低漏模式。保持内部逻辑状态
VLLS3	关闭内核时钟。系统时钟服务其他模块。来自支持外设的停止应答信号后,关闭总线时钟。通过关闭内部逻辑,将MCU置于低漏模式。保存所有系统RAM的内容且保存IO状态。没保存内部逻辑状态
VLLS1	关闭内核时钟。系统时钟服务其他模块。来自支持外设的停止应答信号后,关闭总线时钟。通过关闭内部逻辑和所有系统RAM,将MCU置于低漏模式。保存IO状态。没保存内部逻辑状态
VLSS0	关闭内核时钟。系统时钟服务其他模块。来自支持外设的停止应答信号后,关闭总线时钟。通过关闭内部逻辑和所有系统RAM,将MCU置于低漏模式。保存IO状态。没保存内部逻辑状态。禁用1kHz LPO时钟,使用STOPCTRL[PORPO],可以选择是否开启电源复位电路

每个运行模式都有等待和停止的配合。等待模式对应于ARM的睡眠模式。停止模式(VLPS,STOP)对应于ARM深度睡眠模式。当最大总线频率不是必需的时候,低功耗运行操作模式能最大减少电源消耗。CPU有三种基本模式:运行、等待和停止。WFI指令可以进入等待和停止模式。芯片通过运行、等待和停止三种模式的不同排列来实现低功耗。

RUN模式包含:RUN、VLPR。

WAIT模式包含:WAIT、VLPW。

STOP模式包含:STOP、VLPS、LLS、VLLS3、VLLS1、VLLS0。

对应各个模式在工作电压 3.0V，温度为 25℃时的功耗如表 13-3 所示。

表 13-3　各模式在工作电压 3.0V，温度为 25℃的功耗值

符　　号	描　　述	最小值	典型值	最大值	单位
IDD_RUN	当前处于 RUN 模式，48MHz 内核时钟/24MHz 总线时钟，所有的外设时钟被禁止。执行 Flash 中循环 while(1)里面的代码	—	5.1	6.3	mA
	当前处于 RUN 模式，48 MHz 内核时钟/24 MHz 总线时钟，所有的外设时钟被使能。执行 Flash 中循环 while(1)里面的代码	—	6.4	7.8	mA
IDD_WAIT	当前处于 Wait 模式，内核时钟被禁止/48MHz 系统时钟/24MHz 总线时钟/Flash 被禁止，所有的外设时钟被禁止	—	3.7	5.0	mA
	当前处于 Wait 模式，内核时钟被禁止/24MHz 系统时钟/24MHz 总线时钟/Flash 被禁止，所有的外设时钟被禁止	—	2.9	4.2	mA
IDD_STOP	当前处于 STOP 模式	—	345	490	μA
IDD_VLPR	当前处于 VLPR 模式，4MHz 内核时钟/0.8MHz 总线时钟，所有的外设时钟被禁止。执行 Flash 中循环 while(1)里面的代码	—	224	613	μA
	当前处于 VLPR 模式，4MHz 内核时钟/0.8MHz 总线时钟，所有的外设时钟使能。执行 Flash 中循环 while(1)里面的代码	—	300	745	μA
IDD_VLPW	当前处于 VLPW 模式，内核时钟被禁止/4MHz 系统时钟/0.8MHz 总线时钟/Flash 被禁止，所有的外设时钟被禁止	—	135	496	μA
IDD_VLPS	当前处于 VLPS 模式	—	4.4	16	μA
IDD_LLS	当前处于 LLS 模式	—	1.9	3.7	μA
IDD_VLLS3	当前处于 VLLS3 模式	—	1.4	3.2	μA
IDD_VLLS1	当前处于 VLLS1 模式	—	0.7	1.4	μA
IDD_VLLS0	当前处于 VLLS0 模式，上电复位检测电路使能(SMC_STOPCTRL[PORPO] = 0)	—	956	11 760	μA
	当前处于 VLLS0 模式，上电复位检测电路禁止(SMC_STOPCTRL[PORPO] = 1)	—	760	3577	μA

13.2.2　电源模式转换

电源模式转换可以通过 WFI 指令实现。通过 WFI 可以进入等待和低功耗停止模式(包含 Stop，VLPS，LLS，VLLSx 模式)。芯片通过中断退出低功耗模式。嵌套向量中断控制器(NVIC)描述了中断的不同操作以及哪种外设可以引发中断，退出低功耗模式。

图 13-3 为系统电源模式转换图，任意时刻的芯片复位都会使芯片转到正常的运行状态。在运行、等待和低功耗停止模式的不同转换过程中，必须开启电源调节器的功能。VLPx 模式在频率上受到了限制，但能提供比正常模式更低功耗的工作模式。LLS 和 VLLSx 模式是最低功耗的停止模式，在这些模式下，一些控制逻辑和存储单元的功能被关闭。

图 13-3 系统电源模式转换图

图中各个模式的转换方式,参见KL25参考手册。测试实例参见网上教学资源..\KL25-SMC。

13.3 低漏唤醒单元

1. 低漏唤醒单元(LLWU)模块简介

MCU低功耗系统中的一个关键组件是低漏唤醒单元(Low Leakage Wake Up, LLWU),它在所有低功耗停止模式中充当唤醒监控器。LLWU支持多达16个外部输入引脚(如下降沿、上升沿或可编程的任何方式)和8个可由用户配置的内部外设唤醒事件。

2. 功能说明

允许内部模块和外部输入引脚成为低漏模块唤醒源。它只能在LLS和VLLS模式下工作。LLWU模块包含对每个外部引脚和内部模块的引脚使能。对于每个外部引脚,用户可以禁止或选择唤醒的边沿类型(下降沿、上升沿和边沿触发)。当某个外部引脚作为唤醒源被使能时,此引脚必须被配置为输入引脚。

LLWU实现了一个可选的基于LPO时钟的三周期滤波器,外引脚需保持其引脚状态直到使能滤波器超时,同时还需要有两个周期的延迟,这是因为当滤波器使能时,检测到电路警告系统唤醒或复位事件的同步导致了总共5个周期的延迟。唤醒检测滤波器的实现基于所有使能的外部引脚信号的"或"。内部模块都没有滤波器,对于内部模块的唤醒操作,可通过设置WUMEx位使能对应模块作为唤醒源。

3. LLWU模块特性

(1) 支持多达16个外部引脚唤醒和多达8个内部模块的唤醒源,且拥有独立的使能控制位。

(2) 唤醒源可以是外部引脚或运行于LLS或VLLS模式下的内部外设。

（3）每个外部引脚唤醒输入可以编程为下降沿触发、上升沿触发或边沿触发。

（4）每个内部模块唤醒输入源均有编程使能控制。

（5）一旦使能 MCU 进入低漏模式（LLP），将激活唤醒输入。

（6）一个可选的数字滤波器提供给指定的外部引脚检测，当进入 VLLS0 模式时，过滤器将会进入禁用或者旁路模式。

13.4 看门狗

1. 功能描述

看门狗定时器（WDOG）全称为 Computer Operating Properly（COP）Watchdog，也可以简称 COP。看门狗定时器具有监视系统功能，当运行程序跑飞或一个系统中的关键系统时钟停止引起严重后果的情形下，无法回到正常的程序上执行，看门狗通过复位系统的方式，将系统带到一个安全操作的状态。正常情况下，看门狗通过与软件的定期通信来监视系统的执行过程，将看门狗定时器清零，即定期喂看门狗。如果应用程序丢失，未能在看门狗计数器超时之前清零，则将产生看门狗复位，强制将系统恢复到一个已知的起点。任何复位后，看门狗都将被使能。如果应用程序不使用看门狗，它可以通过 SIM 模块中的 COP 控制寄存器的 SIM_COPC[COPT]位来禁用。

2. 配置 WDOG

1）看门狗计数器复位清零

在正常的工作时间，向 SIM 模块的 SIM_SRVCOP 寄存器，按顺序写入 0x55 和 0xAA 可以软件复位看门狗计数器。写操作不影响在 SIM_SRVCOP 寄存器中的数据，一旦写序列完成后，看门狗计数器将重新计数。如果程序无法在规定的超时时间内执行计数器重置操作，则 MCU 将复位。在超时期间，如果任何 0x55 或 0xAA 以外的值写入 SIM_SRVCOP 寄存器，则 MCU 立即复位。

2）看门狗计数器时钟源选择和超时时间设置

在 SIM 的 COP 控制寄存器 SIM_COPC[COPCLKS]字段中，可以设定用于看门狗定时器的时钟源。可选择的时钟源为总线时钟或内部的 1kHz 时钟源。每种时钟源可以通过 SIM_COPC[COPT]设置三个超时时间。总线时钟源选择后，通过设置在 SIM 的 SIM_COPC[COPW]位来使窗口 COP 操作可用。表 13-4 总结了 COPCLKS、COPT 和 COPW 位的控制功能。

表 13-4　COP 配置选项

SIM_COPC 控制位		时钟源	COP 窗口打开 SIM_COPC[COPW] = 1	COP 溢出计数
COPCLKS	COPT			
—	00	—	—	COP 被禁用
0	01	1kHz	—	2^5 个周期（32ms）
0	10	1kHz	—	2^8 个周期（256ms）
0	11	1kHz	—	2^{10} 个周期（1024ms）

续表

SIM_COPC 控制位		时钟源	COP 窗口打开 SIM_COPC[COPW]=1	COP 溢出计数
COPCLKS	COPT			
1	01	总线	6144 个周期	2^{13}个周期(32ms)
1	10	总线	49 152 个周期	2^{16}个周期(32ms)
1	11	总线	196 608 个周期	2^{18}个周期(32ms)

在窗口模式下,写入 SRVCOP 寄存器以清除 COP 定时器的操作,必须发生在选定的超时时间剩余 25%以后,过早地进行写操作将立即复位芯片。当选择 1kHz 时钟源时,COP 窗口操作不可用。

3) 其他说明

第一次对 SIM 模块的 SIM_COPC 寄存器的写入操作或者任何系统复位,都会使 COP 计数器初始化,对 SIM 模块的 SIM_COPC 寄存器的后续的写入操作将不影响 COP 的操作。特别提醒：这表明 MCU 复位之后,SIM_COPC 寄存器只能写入一次,第二次及之后的写入操作无效。

如果选择总线时钟作为时钟源,在 MCU 处于调试模式或者系统处于停止模式(包括 VLPS 或 LLS)时,COP 计数器不会增加。当 MCU 退出调试模式或停止模式时,COP 计数器恢复计时。

如果选择 1 kHz 时钟源时,MCU 一旦进入调试模式或停止模式(包括 VLPS 或 LLS),COP 计数器都将被重新初始化为零。退出调试模式或停止模式后,计数器从零开始计时。无论选择哪种时钟,只要当芯片进入一个 VLLSx 模式,COP 都将被禁止。从 VLLSx 模式唤醒芯片复位后,COP 将重新初始化。

3. 测试实例

测试实例参见随书光盘..\ KL25-COP。在系统开发过程中,一般先关闭看门狗的功能,避免不必要的复位启动事件的发生。只有在系统开发完成,调试正常准备投入使用时,才开启看门狗的功能。

13.5 复位模块

芯片被正确写入程序后,经复位或重新上电后才可启动执行程序。当出现异常时,也可通过复位,使得芯片恢复到最初已知状态,以对系统进行保护。KL25 支持的复位源见表 13-5。

表 13-5　复位源

复位源	描　　述
上电复位	上电复位(POR)
系统复位	外部引脚复位(RESET);低电平检测(LVD)复位;COP 看门狗复位;低漏唤醒(LLWU)复位;多用途时钟发生器时钟丢失(LOC)复位;多用途时钟发生器失锁(LOL)复位;停止模式应答错误(SACKERR)复位;软件复位;锁定复位(LOCKUP);MDM-AP 系统复位
调试复位	复位调试子系统

每个系统复位源对应系统复位状态寄存器(RCM_SRS)里的相应位。下面介绍下不同复位的复位条件。

13.5.1　上电复位

当给 MCU 上电或提供的电压低于上电复位重置电压(VPOR)时,POR 电路会触发 POR 复位。当电压升高时,低电压检测(LVD)电路保持 MCU 处于复位状态直到电压大于 LVD 低电压阈值(VLVDL)。POR 复位后 SRSL 寄存器的 POR 和 LVD 位也要重置。

13.5.2　系统复位源

MCU 复位是一种可以使芯片回到初始状态的方法。系统复位由片上稳压器监控,系统时钟由内部参考时钟产生。当芯片退出复位时,它执行以下操作。

(1) 从中断向量表偏移 0 读取起始 SP(SP_main)。

(2) 从中断向量表偏移 4 读取起始 PC。

(3) LR 设置为 0xFFFF_FFFF。

片上外设模块和非模拟 IO 引脚初始化都被置为禁用。复位之后模拟引脚被默认为相应的模拟功能。复位时或者复位之后,SWD 相应的输入引脚被配置为:SWD_CLK 上拉,SWD_DIO 下拉。

1. 外部引脚复位(RESET)

该引脚被定义为开漏和内部上拉。该位被置后,将芯片从任何模式唤醒。通过置位 RESET_PIN_CFG 选项为 0,复位引脚可以被禁用。

2. 低电平检测(LVD)复位

当给芯片提供多种电压,芯片的低电平检测系统可以保护内存内容和控制 MCU 系统状态。系统由上电复位电路(POR)和低电压检测(LVD)电路组成。对于低电压检测(LVD)电路,用户可选择电压(高电压(VLVDH)或低电压(VLVDL))。通过设置 PCM 的 LVDSC1[LVDRE]位为 1,LVD 可以在检测到低电压条件时产生复位。当产生 LVD 复位时,LVD 系统使 MCU 处于复位状态直到供应的电压大于 LVD 低电压阈值。LVD 复位或 POR 时 SRSL[LVD]位被置位。

3. COP 看门狗复位

看门狗定时器通过软件周期性的通信操作监视系统的操作。该通信通常称为服务性看门狗。如果没有定时进行喂狗,看门狗将产生系统复位。COP 复位导致 SRS0[WDOG]位被置位。

4. 低漏唤醒(LLWU)复位

LLWU 模块为用户提供了用外部引脚和内部外设将 MCU 从低漏模式下唤醒。LLWU 模块只有在低漏功耗模式下使用。在 VLLSx 模式下,所有使能的输入 LLWU 都可以产生系统复位。系统复位后,LLWU 会保持标识上一次的唤醒源的标志直到用户清该标志位。LLWU 复位中外设模块的一些条件标志位会自动被清除。

5. 多功能时钟发生器时钟丢失(LOC)复位

如果 MCG 中的 C6[CME0]为 1,时钟监视器使能。当外部参考频率低于 f_{loc_low} 或者 f_{loc_high}(由 MCG 中的 C2[RANGE0]位控制),MCU 复位。复位控制模块的 SRS0[LOC]位

为1代表该复位源。

6. 多用途时钟发生器失锁(LOL)复位

MCG 有一个 PLL 丢失锁定探测器。当 MCG 配置为 PEE 或者锁定使能,该探测器使能。如果 MCG_C8[LOLRE]位被置位并且 PLL 锁的状态位(MCG_S[LOLS0])变成 1,MCU 复位。RCM_SRS0[LOL]位标志该复位源。如果芯片在任何停止模式下,该复位源不会产生任何复位。

7. 停止模式应答错误(SACKERR)复位

如果内核试图进入停止模式或者计算操作,该复位产生,但是不是所有模块在 1kHz LPO 时钟的 1025 个周期内产生停止模式应答。如果一个错误产生,该模块可能并不应答进入停止模式。该复位也可能由外部时钟输入模块中产生。

8. 软件复位(SW)

NVIC 应用中断和复位控制寄存器的 SYSRESETREQ 位被置位时会产生一个软件复位。置位 SYSRESETREQ 可产生软件复位请求。除了调试模块,软件复位能重设系统大多数模块。软件复位时 RCM 的 SRS1[SW]位被置位。

9. 锁定复位(LOCKUP)

LOCKUP 会立即表明内核软件产生严重的错误。在内核中系统硬件保护激活时,一个不可恢复异常导致内核被锁。LOCKUP 导致系统复位,也使 RCM 的 SRS1[LOCKUP]位被置位。

10. MDM-AP 系统复位

通过设置 MDM-AP 控制寄存器中的系统复位请求位可以产生系统复位。此复位方式是 SWD 接口的主要复位方法。直到该位被清,系统复位才停止。当芯片从系统复位中唤醒,通过设置 MDM-AP 控制寄存器中的内核保持复位可以使内核保持复位状态。

13.5.3　调试复位

使用 DP CTRL/STAT 寄存器的 CDBGRSTREQ 位复位调试模块。但是使用 CDBGRSTREQ 位并没有复位所有调试相关的寄存器。

13.6　位操作引擎技术及应用方法

13.6.1　位操作引擎概述

位操作引擎(Bit Manipulation Engine,BME)对基于 Cortex-M0+微控制器的外设地址空间内存的原子"读-改-写"操作提供了硬件支持。为外设地址数据的读写操作提供了一种封装地址方案,BME 的封装参数可以从系统总线事务中获得,由处理器内核产生。

BME 提供的封装式存储包含三个逻辑操作(AND、OR、XOR)和一个数据位插入操作。对于这些操作来说,BME 是把一个单独的封装式 AHB 存储事务处理过程转换成一个包含两个周期的原子级"读-改-写"操作序列。在第一个 AHB 数据序列中包含读-改操作,在第二个数据序列中包含写操作。

BME 提供的封装式载入有如下功能：两个单独的置位和清除操作、无符号数据位提取操作。对于置位和清除操作来说，BME 是把一个单独的封装式 AHB 存储事务处理过程转换成一个包含两个周期的原子级“读-改-写”操作序列，在第一个 AHB 数据序列中包含读-改操作，在第二个数据序列中包含写操作。然后读出的原始数据返回到处理器内核中。对无符号位提取操作来说，在数据被提取时，封装式载入事务处理操作是停滞的。在第二个 AHB 数据序列中，返回到处理器。该操作只是一个简单的数据读取操作，不是读-改-写操作。

BME 提供的封装式存储和载入不同功能所对应的外设地址的各个位含义如表 13-6 所示。其中，AND、OR、XOR、BFI 4 种操作为封装地址写操作；LAC1、LAS1、UBFX 为封装地址读操作。

表 13-6　封装式存储和载入功能外设地址各位含义表

<table>
<tr><th>功　　能</th><th>位 31</th><th>位 30、29</th><th>位 28</th><th>位 27、26</th><th>位 25～21</th><th>位 20</th><th>位 19～0</th></tr>
<tr><td>逻辑与 AND</td><td>0</td><td rowspan="5">10</td><td>0</td><td>01</td><td rowspan="3">未使用</td><td rowspan="5">未使用</td><td rowspan="5">SRAM _U 或外设偏移地址</td></tr>
<tr><td>逻辑或 OR</td><td>0</td><td>0</td><td>10</td></tr>
<tr><td>逻辑异或 XOR</td><td>0</td><td>0</td><td>11</td></tr>
<tr><td>清除一位 LAC1</td><td>0</td><td>0</td><td>10</td><td rowspan="2">b：位标识符</td></tr>
<tr><td>置位 LAS1</td><td>0</td><td>0</td><td>11</td></tr>
<tr><td>功能</td><td>位 31</td><td>位 30、29</td><td>位 28</td><td colspan="2">位 27～23</td><td>位 22～19</td><td>位 18～0</td></tr>
<tr><td>数据位插入 BFI</td><td>0</td><td rowspan="2">10</td><td>1</td><td colspan="2" rowspan="2">b：位标识符</td><td rowspan="2">w：位宽度</td><td rowspan="2">SRAM _U 或外设偏移地址</td></tr>
<tr><td>无符号位提取 UBFX</td><td>0</td><td>1</td></tr>
</table>

根据 KL25 参考手册得知，外设地址空间占了 516KB：基址是 0x4000_0000 的 512KB 准外设地址空间和基址是 0x400F_F000 的 4KB GPIO 访问空间。封装地址空间是一个 448MB 的区域，范围为 0x4400_0000～0x5FFF_FFFF。与 SRAM_U 关联的封装地址空间也是一个 448MB 的区域，范围为 0x2400_0000～0x3FFF-FFFF。

13.6.2　位操作引擎的应用机制解析

下面介绍利用位操作引擎 BME 技术对外设地址的操作。

(1) 位操作引擎操作外设地址空间。

根据手册得知，KL25 微控制器 PORTB19 是 GPIOB 寄存器的第 19 位，其地址为 *(volatile unsigned long int *)(unsigned long int) 0x400FF000。我们以 PORTB19 为例，将输出寄存器的第 19 位清 0，使 PORTB19 引脚输出低电平。可以利用清除一位操作方法和 AND 操作方法。

(2) 通常位操作方法的基本过程分析——“读-改-写”。

通常所说的“读-改-写”操作实现对外设地址空间的访问操作。

① 读一个字：读出 0x4400_0000～0x5FFF_FFFF 中内容到变量 temp 中。

```
temp=( * ( volatile unsigned long int * )(unsigned long int) 0x4400FF00);
```

② 改一个位：将 temp 中的第 19 位清 0。

```
temp=temp&(0xFFF7FFFF);
```

③ 写一个字：将temp写回目标地址。

```
(*(volatile unsigned long int *)(unsigned long int) 0x4400FF00)=temp;
```

这就是通常所说的“读-改-写”操作，即读内存赋给临时变量，然后对临时变量进行修改，最后将临时变量结果写回内存。

(3) 利用BME的清除一位操作方法。根据表13-6可以编写对应外设封装地址：

位31	位30、29	位28	位27、26	位25～21	位20	位19～0
0	10	0	10	10011	0	1111 1111 0000 0000 0000

```
addr[31:0]=0100 1010 0110 1111 1111 0000 0000 0000=0x4A6FF000
U32temp=(*( volatile unsigned long int *)(unsigned long int) 0x4A6FF000);
```

读取该外设封装地址，即完成了清除一位操作。

(4) 利用BME的AND操作操作方法。根据表13-6可以编写对应外设封装地址：

位31	位30、29	位28	位27、26	位25～21	位20	位19～0
0	10	0	01	00000	0	1111 1111 0000 0000 0000

```
addr[31:0]=0100 0100 0000 0000 0010 1111 1111 0000=0x440FF000
(*( volatile unsigned long int *)(unsigned long int) 0x440FF000)= 0xFFF7FFFF;
```

写0xFFF7FFFF给该外设封装地址，即完成与运算操作，实现第19位清0。

(5) 分析。

我们在KDS3.0.0编译环境中对上述代码反汇编进行对比，可以发现使用BME技术比原有“读-改-写”方法的代码空间要小，执行效率更高。

一般情况下的机器码如下。

```
//*****************************************************************
//外设地址读一个字
 temp=(*( volatile unsigned long int *)(unsigned long int)0x400FF000);
 850:    4b2a          ldr    r3, [pc, #168]    ; (8fc <main+0xfc>)
 852:    681b          ldr    r3, [r3, #0]
 854:    60fb          str    r3, [r7, #12]
    //改一个位
    temp=temp&(0xFFF7FFFF);
 856:    68fa          ldr    r2, [r7, #12]
 858:    4b24          ldr    r3, [pc, #144]    ; (8ec <main+0xec>)
 85a:    4013          ands    r3, r2
 85c:    60fb          str    r3, [r7, #12]
    //外设地址写一个字
    (*( volatile unsigned long int *)(unsigned long int)0x400FF000)=temp;
 85e:    4b27          ldr    r3, [pc, #156]    ; (8fc <main+0xfc>)
 860:    68fa          ldr    r2, [r7, #12]
 862:    601a          str    r2, [r3, #0]
```

使用 BME 方法的机器码如下。

```
//BME 操作外设寄存器
//BME 清除一位
 U32temp=(*( volatile unsigned long int *)(unsigned long int) 0x4A6FF000);
 864:    4b26          ldr    r3, [pc, #152]     ; (900 <main+0x100>)
 866:    681b          ldr    r3, [r3, #0]
 868:    60bb          str    r3, [r7, #8]
    //BME 的 AND 操作
    (*( volatile unsigned long int *)(unsigned long int) 0x440FF000)= 0xFFF7FFFF;
 86a:    4b26          ldr    r3, [pc, #152]     ; (904 <main+0x104>)
 86c:    4a1f          ldr    r2, [pc, #124]     ; (8ec <main+0xec>)
 86e:    601a          str    r2, [r3, #0]
```

13.6.3　位操作引擎对 GPIO 部分的使用说明

外设地址空间占用 516KB：512KB 基于 0x4000_0000 加上 4KB 基于 0x400F_F000 空间的 GPIO 访问。这种内存布局主要是为了兼容 Kineti K 系列 MCU。

GPIO 地址空间被硬件多重映射。它出现在“标准”系统地址 0x400F_F000 和在物理上位于与槽对应的地址为 0x4000_F000。使用 addr[19]进行封装操作。对于 AND,OR,XOR,LAC1 和 LAS1,这个位作为真实的地址位；对于 BFI 和 UBFX,这个位定义了 w 字段说明符的最低有效位。

因此,直接对 GPIO 的访问操作和封装的 AND,OR,XOR,LAC1 和 LAS1 操作可以使用标准 0x400F_F000 基地址,然而 BFI 和 UBFX 封装操作必须轮流使用 0x400F_F000 基地址。当然可以简单地使用 0x400F_F000 作为所有未封装的 GPIO 访问和把 0x4000_F000 作为所有封装访问的基地址。UBFX 和 BFI 一样操作的都是映射内存空间,用来操作 GPIO 时要以 F000 为起始地址。具体访问说明见表 13-7。

表 13-7　外设封装和 GPIO 地址细节

外设地址空间	说　明
0x4000_0000～0x4007_FFFF	未封装(正常)外设访问
0x4008_0000～0x400F_EFFF	非法地址；试图引用会失败并错误终止
0x400F_0000～0x400F_FFFF	使用标准地址进行未封装(正常)GPIO 访问
0x4010_0000～0x43FF_FFFF	非法地址；试图引用会失败并错误终止
0x4400_0000～0x4FFF_FFFF	封装 AND,OR,XOR,LAC1,LAS1 引用外设,GPIO 基于 0x4000_F000 或 0x400F_F000
0x5000_0000～0x5FFF_FFFF	封装 BFI,UBFX 引用外设,GPIO 只基于 0x4000_F000

13.6.4　位操作引擎使用注意点

1. 数据位宽度

上述范例中仅以 32 位字的操作为例进行了说明,BME 对于各种操作均支持 8、16、32 位宽度的数据。

2. w1c 位的操作

BME 执行的是读-修改-写操作，而很多寄存器有些位是 w1c，也就是所谓的 write-1-clear，写 1 清 0 的工作方式。使用 BME 时就需要特别注意和小心了，否则会出现很多不可预料的后果。如果一个寄存器中有多个连续的 w1c 位，就不要使用 LAS1 来对寄存器写 1 清 0 了，因为在 LAS1 这个操作中，其中有一步操作是将数据读回(在 KL25 参考手册中提到“read data return to core”)。这一步会将原本不需要清 0 的位给清了。所以 BME 用于处理 w1c 位时要特别小心。

3. 注意使用 volatile 关键字

同样在使用 BME 技术时，所访问的存储器单元变量也必须使用关键字 volatile 来加以定义。

13.6.5　测试实例

测试工程位于网上教学资源中的“..\ KL25_Bit-Band&BME”文件夹。功能是利用多种方法对 KL25 板上三色灯中红灯所对应的 GPIO 引脚控制。此外，可以通过 KDS3.0.0 在线调试方法，观测利用位带技术和 BME 技术对外设区的操作过程。

小　　结

本章主要阐述了 KL25 系列芯片的时钟系统，时钟源、时钟发生器 MCG、时钟信号的设置和选择；分析了电源管理模块、低功耗的各种状态的切换、低功耗状态的唤醒；介绍了看门狗模块和系统的复位和启动。本章的内容需读者全面理解，掌握时钟模块的结构组成，设置、选择和使用各种时钟信号。

(1) 详细分析时钟系统的结构组成原理，KL25 芯片的时钟系统由振荡器(OSC)、实时时钟(RTC)、多功能时钟发生器(MCG)、系统集成模块(SIM)和电源管理器(PMC)等模块组成；时钟源可以来自外部晶振提供的参考时钟，也可以来自内部参考时钟；分析了时钟信号的产生是通过 MCG 模块来控制和编程的，而系统的时钟分频器和模块时钟门是通过 SIM 模块来编程设置，为其他模块提供时钟信号。分析各时钟信号寄存器的设置方法，并给出了时钟模块测试实例。

(2) 分析了 CPU 有三种基本模式下的电源管理，在不同模式下电源可进入不同的状态，从而达到降低功耗的目的。RUN 模式包含 RUN、VLPR；WAIT 模式包含 WAIT、VLPW；STOP 模式包含 STOP、VLPS、LLS、VLLS3、VLLS1、VLLS0。介绍了 MCU 低功耗系统中的一个关键组件是低漏唤醒单元(LLWU)，它在所有低功耗停止模式中充当唤醒监控器。

(3) 介绍了低漏唤醒单元，位带操作。这些模块涉及嵌入式系统的基础内容，虽然不会经常显式地用到，但也是嵌入式开发过程中必备的知识。

(4) 介绍了看门狗模块。在系统开发过程中，一般先关闭看门狗功能，避免不必要复位的发生。只有在系统开发完成，调试正常准备投入使用时，才开启看门狗的功能。规范地使用看门狗可以有效地防止程序跑飞。

（5）本章还介绍了复位模块。介绍了不同的复位源以及各个复位发生的条件。复位模块可以在出现异常时使得芯片恢复到最初已知状态，以对系统进行保护。

习　　题

1. 简要阐述 KL25 系列芯片各个模块使用的时钟类型。

2. 简述 KL25 芯片 ADC 构件中的时钟配置。

3. 简述 KL25 芯片电源模式的种类及各自的特点。

4. 简述 KL25 芯片的复位源的类别，并设计一个程序，识别程序是哪种复位。

5. 请编写程序，实现功能：

（1）看门狗定时器的超时时间为 1s；

（2）添加看门狗定时器中断函数，增加一个计数器并递增计数器值，在主程序中输出看门狗复位次数。

第14章　进一步学习指导

14.1　关于更为详细的技术资料

本书作为教材，通用知识占用一部分篇幅。ARM 及 Freescale 提供的参考手册、数据手册等材料比较多，见参考文献[1]～[12]，电子文档在本书网上教学资源中，可以参阅。

14.2　关于实时操作系统 RTOS

实时操作系统 RTOS 是嵌入式系统学习的重要内容之一。关于 RTOS，针对 ARM Cortex-M 系列微处理器，推荐使用开源嵌入式实时操作系统 MQX。有关内容在《嵌入式实时操作系统 MQX 应用开发技术》一书中阐述。这里简要给出什么是 RTOS、何时使用 RTOS、如何选择 RTOS 以及选择 MQX 的理由。

1. 什么是实时操作系统

操作系统（Operating System，OS）是一套管理计算机硬件与软件资源的程序，是计算机的系统软件。一般 PC 操作系统提供设备驱动管理、进程管理、存储管理、文件系统、安全机制、网络通信及使用者界面等功能。

嵌入式操作系统（Embedded Operation System，EOS）是相对于一般操作系统而言的，是一种工作在嵌入式计算机系统上的系统程序。一般情况下，它嵌入到微控制器、应用处理器或其他存储载体中。它与一般操作系统最基本的功能类似，EOS 负责嵌入系统的软、硬件资源的分配、任务调度、同步机制、中断处理等功能。

嵌入式实时操作系统（Embedded Real Time Operation System，ERTOS，一般直接简称 RTOS）是一种具有较高实时性的嵌入式操作系统。实时是指能够在确定的时间内完成特定的系统功能或中断响应。

在无 RTOS 的嵌入式系统中，系统复位后，首先进行堆栈、系统时钟、内存变量、部分硬件模块、中断等初始化工作，然后进入"永久循环"。在这个循环中，CPU 顺序执行各种功能程序（任务），这是一条运行路线。若发生中断，将响应中断，执行中断服务例程（Interrupt Service Routines，ISR），这是另一条运行路线，执行完 ISR 后，返回中断处继续执行。

从操作系统功能角度理解，上述程序可以看作是一个 RTOS 内核，这个内核负责系统初始化和调度其他任务。RTOS 就是这样的一个标准内核，包括芯片初始化、设备驱动及数据结构的格式化，应用层程序员可以不对硬件设备和资源进行直接操作，而是通过标准调用方法实现对硬件的访问，所有的任务由 RTOS 内核负责调度。

一个典型 RTOS 的特征：①允许多任务；②带有优先级的任务调度；③资源的同步访

问；④任务间的通信；⑤定时时钟；⑥中断处理。

2. 何时使用 RTOS

首先考虑系统是否复杂到一定需要用一个 RTOS，且其硬件又具备足够的处理能力时，或系统当前功能及将来可能需要扩展，则考虑使用 RTOS。具体来讲：

(1) 需要并行运行多个较复杂的任务，任务间需要进行实时交互；

(2) 需要为应用程序提供统一的 API，实现应用软件与硬件驱动独立开发，便于应用程序的开发与维护；

(3) 需要开发硬件相似但功能不同的产品，代码能方便地移植和复用。

3. 如何选择 RTOS

可以从性能、技术支持与成本、资源等角度进行考虑：

(1) 性能如何？内核要求的最小开销；以及可维护性、可移植性、可扩展性。

(2) 技术支持如何？是否免费、是否有版税、是否可以深度开发、是否有收费陷阱等。

(3) 相关工具的考虑，如微处理器、在线仿真器、编译器、汇编器、连接器、调试器以及模拟器等工具是否成熟；是否提供驱动和应用程序库；是否提供驱动及中间件(如 USB、GUI、以太网、Wi-Fi、文件系统、传感器、安全等)。

4. 选择 MQX 的原因

MQX 已经走过了 15 年的发展历程，被广泛应用于医疗电子、工业控制等领域，基于 MQX 的产品已达数百万。

(1) 实时性高，提供高效的任务调度、内存管理等功能；系统精简，代码最小 16KB，RAM 最小开销 2KB，对硬件系统开销较小。

(2) MQX 内核完全免费；由 Freescale 公司团队提供技术支持、升级；同时不断推出新系列芯片的驱动。

(3) 支持 Codewarrior，Keil 和 IAR，工具成熟，上手快；提供丰富的驱动、中间件和应用程序库，这使得用户更加关注于他们需要的功能上，而非 MQX 的堆栈、驱动等；恩智浦提供免费 MQX RTOS、USB、TCP/IP 和 MFS 协议栈，降低了开发成本。

(4) 与 Linux 相比，Linux 的 MMU、OpenGL 功能强大，占用资源多，但 MQX 内核精简，实时性强、效率高，更适合于医疗电子、工业控制等领域。与 μCOS 相比，核心大小接近，但 MQX 的维护团队强大，提供了众多的驱动，方面用户使用。

14.3　关于嵌入式系统稳定性问题

看到这里，读者基本上具备了进行嵌入式系统开发的软硬件基础，但是实际开发嵌入式产品时远不止于此。稳定性是嵌入式系统的生命线，而实验室中的嵌入式产品在调试、测试、安装之后，最终投放到实际应用，往往还会出现很多故障和不稳定的现象。由于嵌入式系统是一个综合了软件和硬件的复杂系统，因此仅依靠哪个方面都不能完全解决其抗干扰问题，只有从嵌入式系统硬件、软件以及结构设计等方面进行全面的考虑，综合应用各种抗干扰技术来全面应对系统内外的各种干扰，才能有效提高其抗干扰性能。在这里，作者根据多年来的嵌入式产品开发经验，对实际项目中较常出现的稳定性问题做简要阐述，供读者在

进一步学习中参考。

嵌入式系统的抗干扰设计主要包括硬件和软件两个方面。在硬件方面通过提高硬件的性能和功能,能有效地抑制干扰源,阻断干扰的传输信道,这种方法具有稳定、快捷等优点,但会使成本增加。而软件抗干扰设计采用各种软件方法,通过技术手段来增强系统的输入输出、数据采集、程序运行、数据安全等抗干扰能力,具有设计灵活、节省硬件资源、低成本、高系统效能等优点,且能够处理某些用硬件无法解决的干扰问题。

1. 保证 CPU 运行的稳定

CPU 指令由操作码和操作数两部分组成,取指令时先取操作码后取操作数。当程序计数器 PC 因干扰出错时,程序便会跑飞,引起程序混乱失控,严重时会导致程序陷入死循环或者误操作。为了避免这样的错误发生或者从错误中恢复,通常使用指令冗余、软件拦截技术、数据保护、计算机操作正常监控(看门狗)和定期自动复位系统等方法。

2. 保证通信的稳定

在嵌入式系统中,会使用各种各样的通信接口,以便与外界进行交互,因此,必须要保证通信的稳定。在设计通信接口的时候,通常从通信数据速度、通信距离等方面进行考虑,一般情况下,通信距离越短越稳定,通信速率越低越稳定。例如,对于 UART 接口,通常只选用 9600、38 400、115 200 等低速波特率来保证通信的稳定性,另外,对于板内通信,使用 TTL 电平即可,而板间通信通常采用 232 电平,有时为了传输距离更远,可以采用差分信号进行传输。

另外,通过为数据增加校验也是增强通信的稳定性的常用方法,甚至有些校验方法不仅具有检错功能,还具有纠错功能。常用的校验方法有奇偶校验、循环冗余校验法(CRC)、海明码以及求和校验和异或校验等。

3. 保证物理信号输入的稳定

模拟量和开关量都是属于物理信号,它们在传输过程中很容易受到外界的干扰,雷电、可控硅、电机和高频时钟等都有可能成为其干扰源。在硬件上选用高抗干扰性能的元器件可有效地克服干扰,但这种方法通常面临着硬件开销和开发条件的限制。相比之下,在软件上则可使用的方法比较多,且开销低,容易实现较高的系统性能。

通常的做法是进行软件滤波,对于模拟量,主要的滤波方法有限幅滤波法、中位值滤波法、算术平均值法、滑动平均值法、防脉冲干扰平均值法、一阶滞后滤波法以及加权递推平均滤波法等;对于开关量滤波,主要的方法有同态滤波和基于统计计数的判定方法等。

4. 保证物理信号输出的稳定

系统的物理信号输出,通常是通过对相应寄存器的设置来实现的,由于寄存器数据也会因干扰而出错,所以使用合适的办法来保证输出的准确性和合理性也很有必要,主要方法有输出重置、滤波和柔和控制等。

在嵌入式系统中,输出类型的内存数据或输出 I/O 口寄存器也会因为电磁干扰而出错,输出重置是非常有效的办法。定期向输出系统重置参数,这样,即使输出状态被非法更改,也会在很短的时间里得到纠正。但是,使用输出重置需要注意的是,对于某些输出量,如 PWM,短时间内多次的设置会干扰其正常输出。通常采用的办法是,在重置前先判断目标

值是否与现实值相同,只有在不相同的情况下才启动重置。有些嵌入式应用的输出,需要某种程度的柔和控制,可使用前面所介绍的滤波方法来实现。

总之,系统的稳定性关系到整个系统的成败,所以在实际产品的整个开发过程中都必须要予以重视,并通过科学的方法进行解决,这样才能有效地避免不必要的错误的发生,提高产品的可靠性。

附录 A　KL25/26 芯片引脚复用功能

A.1　KL25 引脚复用功能

KL25 引脚复用功能如附表 A-1 所示。

附表 A-1　KL25 引脚复用功能

80 LQFP	64 LQFP	48 QFN	32 QFN	引脚名	默认功能	ALT0	ALT1	ALT2	ALT3	ALT4	ALT5	ALT6	ALT7
1	1	—	1	PTE0	DISABLEO		PTE0		UART1_TX	RTC_CLKOUT	CMP0_OUT	I2C1_SDA	
2	2	—	—	PTE1	DISABLED		PTE1	SPI1_MOSI	UART1_RX		SPI1_MISO	I2C1_SCL	
3	—	—	—	PTE2	DISABLED		PTE2	SPI1_SCK					
4	—	—	—	PTE3	DISABLED		PTE3	SPI1_MISO			SPI1_MOSI		
5	—	—	—	PTE4	DISABLED		PTE4	SPI1_PCSO					
6	—	—	—	PTE5	DISABLED		PTE5						
7	3	1	—	VDD	VDD	VDD							
8	4	2	2	VSS	VSS	VSS							
9	5	3	3	USB0_DP	USB0_DP	USB0_DP							
10	6	4	4	USB0_DM	USB0_DM	USB0_DM							
11	7	5	5	VOUT33	VOUT33	VOUT33							
12	8	6	6	VREGIN	VREGIN	VREGIN							
13	9	7	—	PTE20	ADC0_DP0/ ADC0_SE0	ADC0_DP0/ ADC0_SE0	PTE20		TPM1_CH0	UART0_TX			
14	10	8	—	PTE21	ADC0_DM0/ ADC0_SE4a	ADC0_DM0/ ADC0_SE4a	PTE21		TPM1_CH1	UART0_RX			
15	11	—	—	PTE22	ADC0_DP3/ ADC0_SE3	ADC0_DP3/ ADC0_SE3	PTE22		TPM2_CH0	UART2_TX			
16	12	—	—	PTE23	ADC0_DM3/ ADC0_SE7a	ADC0_DM3/ ADC0_SE7a	PTE23		TPM2_CH1	UART2_RX			
17	13	9	7	VDDA	VDDA	VDDA							
18	14	10	—	VREFH	VREFH	VREFH							
19	15	11	—	VREFL	VREFL	VREFL							
20	16	12	8	VSSA	VSSA	VSSA							
21	17	13	—	PTE29	CMP0_IN5/ ADC0_SE4b	CMP0_IN5/ ADC0_SE4b	PTE29		TPM0_CH2	TPM_CLKIN0			
22	18	14	9	PTE30	DAC0_OUT/ ADC0_SE23/ CMP0_IN4	DAC0_OUT/ ADC0_SE23/ CMP0_IN4	PTE30		TPM0_CH3	TPM_CLKIN1			
23	19	—	—	PTE31	DISABLED		PTE31		TPM0_CH4				
24	20	15	—	PTE24	DISABLED		PTE24		TPM0_CH0		I2C0_SCL		

续表

80 LQFP	64 LQFP	48 QFN	32 QFN	引脚名	默认功能	ALT0	ALT1	ALT2	ALT3	ALT4	ALT5	ALT6	ALT7
25	21	16	—	PTE25	DISABLED		PTE25		TPM0_CH1		I2C0_SDA		
26	22	17	10	PTA0	SWD_CLK	TSI0_CH1	PTA0		TPM0_CH5				SWD_CLK
27	23	18	11	PTA1	DISABLED	TSI0_CH2	PTA1	UART0_RX	TPM2_CH0				
28	24	19	12	PTA2	DISABLED	TSI0_CH3	PTA2	UART0_TX	TPM2_CH1				
29	25	20	13	PTA3	SWD_DIO	TSI0_CH4	PTA3	I2C1_SCL	TPM0_CH0				SWD_DIO
30	26	21	14	PTA4	NMI_b	TSI0_CH5	PTA4	I2C1_SDA	TPM0_CH1				NMI_b
31	27	—	—	PTA5	DISABLED		PTA5	USB_CLKIN	TPM0_CH2				
32	28	—	—	PTA12	DISABLED		PTA12		TPM1_CH0				
33	29	—	—	PTA13	DISABLED		PTA13		TPM1_CH1				
34	—	—	—	PTA14	DISABLED		PTA14	SPI0_PCS0	UART0_TX				
35	—	—	—	PTA15	DISABLED		PTA15	SPI0_SCK	UART0_RX				
36	—	—	—	PTA16	DISABLED		PTA16	SPI0_MOST			SPI0_MISO		
37	—	—	—	PTA17	DISABLED		PTA17	SPI0_MISO			SPI0_MOSI		
38	30	22	15	VDD	VDD	VDD							
39	31	23	16	VSS	VSS	VSS							
40	32	24	17	PTA18	EXTAL0	EXTAL0	PTA18		UART1_RX	TPM_CLKIN0			
41	33	25	18	PTA19	XTAL0	XTAL0	PTA19		UART1_TX	TPM_CLKIN1		LPTMR0_ALT1	
42	34	26	19	RESET_b	RESET_b		PTA20						
43	35	27	20	PTB0/LLWU_P5	ADC0_SE8/ TSI0_CH0	ADC0_SE8/ TSI0_CH0	PTB0/LLWU_P5	I2C0_SCL	TPM1_CH0				
44	36	28	21	PTB1	ADC0_SE9/ TSI0_CH6	ADC_SE9/ TSI0_CH6	PTB1	I2C0_SDA	TPM1_CH1				
45	37	29	—	PTB2	ADC0_SE12/ TSI0_CH7	ADC_SE12/ TSI0_CH7	PTB2	I2C0_SCL	TPM2_CH0				
46	38	30	—	PTB3	ADC0_SE13/ TSI0_CH8	ADC_SE13/ TSI0_CH8	PTB3	I2C0_SDA	TPM2_CH1				
47	—	—	—	PTB8	DISABLED		PTB8		EXTRG_IN				
48	—	—	—	PTB9	DISABLED		PTB9						
49	—	—	—	PTB10	DISABLED		PTB10	SPI1_PCS0					
50	—	—	—	PTB11	DISABLED		PTB11	SPI1_SCK					
51	39	31	—	PTB16	TSI0_CH9	TSI0_CH9	PTB16	SPI1_MOSI	UARTO_RX	TPM_CLKIN0	SPI1_MISO		

续表

80 LQFP	64 LQFP	48 QFN	32 QFN	引脚名	默认功能	ALT0	ALT1	ALT2	ALT3	ALT4	ALT5	ALT6	ALT7
52	40	32	—	PTB17	TSI0_CH10	TSI0_CH10	PTB17	SPI1_MISO	UARTO_TX	TPM_CLKIN1	SPI1_MOSI		
53	41	—	—	PTB18	TSI0_CH11	TSI0_CH11	PTB18		TPM2_CH0				
54	42	—	—	PTB19	TSI0_CH12	STI0_CH12	PTB19		TPM2_CH1				
55	43	33	—	PTC0	ADC0_SE14/ TSI0_CH13	ADC0_SE14/ TSI0_CH13	PTC0		EXTRG_IN		CMP0_OUT		
56	44	34	22	PTC1/LLWU_P6/ RTC_CLKIN	ADC0_SE15/ TSI0_CH14	ADC0_SE15/ TSI0_CH14	PTC1/LLWU_P6/ RTC_CLKIN	I2C1_SCL		TPM0_CH0			
57	45	35	23	PTC2	ADC0_SE11/ TSI0_CH15	ADC0_SE11/ TSI0_CH15	PTC2	I2C1_SDA		TPM0_CH1			
58	46	36	24	PTC3/LLWU_P7	DISABLED		PTC3/LLWU_P7		UART1_RX	TPM0_CH2	CLKOUT		
59	47	—	—	VSS	VSS	VSS							
60	48	—	—	VDD	VDD	VDD							
61	49	37	25	PTC4/LLWU_P8	DISABLED		PTC4/LLWU_P8	SPI0_PCS0	UART1_TX	TPM0_CH3			
62	50	38	26	PTC5/LLWU_P9	DISABLED		PTC5/LLWU_P9	SPI0_SCK	LPTMR0_ALT2			CMP0_OUT	
63	51	39	27	PTC6/LLWU_P10	CMP0_IN0	CMP0_IN0	PTC6/LLWU_P10	SPI0_MOSI	EXTRG_IN		SPI0_MISO		
64	52	40	28	PTC7	CMP0_IN1	CMP0_IN1	PTC7	SPI0_MISO			SPI0_MOSI		
65	53	—	—	PTC8	CMP0_IN2	CMP0_IN2	PTC8	I2C0_SCL	TPM0_CH4				
66	54	—	—	PTC9	CMP0_IN3	CMP0_IN3	PTC9	I2C0_SDA	TPM0_CH5				
67	55	—	—	PTC10	DISABLED		PTC10	I2C1_SCL					
68	56	—	—	PTC11	DISABLED		PTC11	I2C1_SDA					
69	—	—	—	PTC12	DISABLED		PTC12			TPM_CLKIN0			
70	—	—	—	PTC13	DISABLED		PTC13			TPM_CLKIN1			
71	—	—	—	PTC16	DISABLED		PTC16						
72	—	—	—	PTC17	DISABLED		PTC17						
73	57	41	—	PTD0	DISABLED		PTD0	SPI0_PCS0		TPM0_CH0			
74	58	42	—	PTD1	ADC0_SE5b	ADC0_SE5b	PTD1	SPI0_SCK		TPM0_CH1			
75	59	43	—	PTD2	DISABLED		PTD2	SPI0_MOSI	UART2_RX	TPM0_CH2	SPI0_MISO		
76	60	44	—	PTD3	DISABLED		PTD3	SPI0_MISO	UART2_TX	TPM0_CH3	SPI0_MOSI		
77	61	45	29	PTD4/LLWU_P14	DISABLED		PTD4/LLWU_P14	SPI1_PCS0	UART2_RX	TPM0_CH4			
78	62	46	30	PTD5	ADC0_SE6b	ADC0_SE6b	PTD5	SPI1_SCK	UART2_TX	TPM0_CH5			
79	63	47	31	PTD6/LLWU_P15	ADC0_SE7b	ADC0_SE7b	PTD6/LLWU_P15	SPI1_MOSI	UART0_RX		SPI1_MISO		
80	64	48	32	PTD7	DISABLED		PTD7	SPI1_MISO	UART0_TX		SPI1_MOSI		

A.2　KL26 引脚复用功能

KL26 引脚复用功能如附表 A-2 所示。

附表 A-2　KL26 引脚复用功能

121 BGA	100 LQFP	64 BGA	64 LQFP	48 QFN	32 QFN	引脚名	默认功能	ALT0	ALT1	ALT2	ALT3	ALT4	ALT5	ALT6	ALT7
E4	1	A1	1	—	1	PTE0	DISABLED		PTE0	SPI1_MISO	UART1_TX	RTC_CLKOUT	CNP0_OUT	I2C1_SDA	
E3	2	B1	2	—	—	PTE1	DISABLED		PTE1	SPI1_MOSI	UART1_RX		SPI1_MISO	I2C1_SCL	
E2	3	—	—	—	—	PTE2	DISABLED		PTE2	SPI1_SCK					
F4	4	—	—	—	—	PTE3	DISABLED		PTE3	SPI1_MISO			SPI1_MOSI		
H7	5	—	—	—	—	PTE4	DISABLED		PTE4	SPI1_PCS0					
G4	6	—	—	—	—	PTE5	DISABLED		PTE5						
F3	7	—	—	—	—	PTE6	DISABLED		PTE6			I2S0_MCLK	audioUSB_SOF_OUT		
E6	8	—	3	1	—	VDD	VDD	VDD							
G7	9	C4	4	2	2	VSS	VSS	VSS							
L6	—	—	—	—	—	VSS	VSS	VSS							
F1	10	E1	5	3	3	USB0_DP	USB0_DP	USB0_DP							
F2	11	D1	6	4	4	USB0_DM	USB0_DM	USB0_DM							
G1	12	E2	7	5	5	VOUT33	VOUT33	VOUT33							
G2	13	D2	8	6	6	VREGIN	VREGIN	VREGIN							
H1	14	—	—	—	—	PTE16	ADC0_DP1/ADC0_SE1	ADC0_DP1/ADC0_SE1	PTE16	SPI0_PCS0	UART2_TX	TPM_CLKIN0			
H2	15	—	—	—	—	PTE17	ADC0_DM1/ADC0_SE5a	ADC0_DM1/ADC0_SE5a	PTE17	SPI0_SCK	UART2_RX	TPM_CLKIN1		LPTMR0_ALT3	
J1	16	—	—	—	—	PTE18	ADC0_DP2/ADC0_SE2	ADC0_DP2/ADC0_SE2	PTE18	SPI0_MOSI		I2C0_SDA	SPI0_MISO		
J2	17	—	—	—	—	PTE19	ADC0_DM2/ADC0_SE6a	ADC0_DM2/ADC0_SE6a	PTE19	SPI0_MISO		I2C0_SCL	SPI0_MOSI		
K1	18	G1	9	7	—	PTE20	ADC0_DP0/ADC0_SE0	ADC0_DP0/ADC0_SE0	PTE20		TPM1_CH0	UART0_TX			
K2	19	F1	10	8	—	PTE21	ADC0_DM0/ADC0_SE4a	ADC0_DM0/ADC0_SE4a	PTE21		TPM1_CH1	UART0_RX			
L1	20	G2	11	—	—	PTE22	ADC0_DP3/ADC0_SE3	ADC0_DP3/ADC0_SE3	PTE22		TPM2_CH0	UART2_TX			
L2	21	F2	12	—	—	PTE23	ADC0_DM3/ADC0_SE7a	ADC0_DM3/ADC0_SE7a	PTE23		TPM2_CH1	UART2_RX			
F5	22	F4	13	9	7	VDDA	VDDA	VDDA							

续表

121 BGA	100 LQFP	64 BGA	64 LQFP	48 QFN	32 QFN	引脚名	默认功能	ALT0	ALT1	ALT2	ALT3	ALT4	ALT5	ALT6	ALT7
G5	23	G4	14	10	—	VREFH	VREFH	VREFH							
G6	24	G3	15	11	—	VREFL	VREFL	VREFL							
F6	25	F3	16	12	8	VSSA	VSSA	VSSA							
L3	26	H1	17	13	—	PTE29	CMP0_IN5/ ADC0_SE4b	CMP0_IN5/ ADC0_SE4b	PTE29		TPM0_CH2	TPM_CLKIN0			
K5	27	H2	18	14	9	PTE30	DAC0_OUT/ ADC0_SE23/ CMP0_IN4	DAC0_OUT/ ADC0_SE23/ CMP0_IN4	PTE30		TPM0_CH3	TPM_CLKIN1			
L4	28	H3	19	—	—	PTE31	DISABLED		PTE31		TPM0_CH4				
L5	29	—	—	—	—	VSS	VSS	VSS							
K6	30	—	—	—	—	VDD	VDD	VDD							
H5	31	H4	20	15	—	PTE24	DISABLED		PTE24		TPM0_CH0		I2C0_SCL		
J5	32	H5	21	15	—	PTE25	DISABLED		PTE25		TPM0_CH1		I2C0_SDA		
H6	33	—	—	—	—	PTE26	IDSABLED		PTE26		TPM0_CH5			RTC_CLKOUT	USB_CLKIN
J6	34	D3	22	17	10	PTA0	SWD_CLK	TSI0_CH1	PTA0		TPM0_CH5				SWD_CLK
H8	35	D4	23	18	11	PTA1	DISABLED	TSI0_CH2	PTA1	UART0_RX	TPM2_CH0				
J7	36	E5	24	19	12	PTA2	DISABLED	TSI0_CH3	PTA2	UART0_TX	TPM2_CH1				
H9	37	D5	25	20	13	PTA3	SWD_DIO	TSI0_CH4	PTA3	I2C1_SCL	TPM0_CH0				SWD_DIO
J8	38	G5	26	21	14	PTA4	NMI_b	TSI0_CH5	PTA4	I2C1_SDA	TPM0_CH1				NMI_b
K7	39	F5	27	—	—	PTA5	DISABLED		PTA5	USB_CLKIN	TPM0_CH2			I2S0_TX_BCLK	
E5	—	—	—	—	—	VDD	VDD	VDD							
G3	—	—	—	—	—	VSS	VSS	VSS							
K3	40	—	—	—	—	PTA6	DISABLED		PTA6		TPM0_CH3				
H4	41	—	—	—	—	PTA7	DISABLED		PTA7		TPM0_CH4				
K8	42	H6	28	—	—	PTA12	DISABLED		PTA12		TPM1_CH0			I2S0_TXD0	
L8	43	G6	29	—	—	PTA13	DISABLED		PTA13		TPM1_CH1			I2S0_TX_FS	
K9	44	—	—	—	—	PTA14	DISABLED		PTA14	SPI0_PCS0	UART0_TX			I2S0_RX_CLK	I2S0_TXD0
L9	45	—	—	—	—	PTA15	DISABLED		PTA15	SPI0_SCK	UART0_RX			I2S0_RXD0	
J10	48	—	—	—	—	PTA16	DISABLED		PTA16	SPI0_MOSI			SPI0_MISO	I2S0_RX_FS	I2S0_RXD0
H10	47	—	—	—	—	PTA17	DISABLED		PTA17	SPI0_MISO			SPI0_MOSI	I2S0_MCLK	
L10	48	G7	30	22	15	VDD	VDD	VDD							
K10	49	H7	31	23	16	VSS	VSS	VSS							
L11	50	H8	32	24	17	PTA18	EXTAL0	EXTAL0	PTA18		UART1_RX	TPM_CLKIN0			

续表

121 BGA	100 LQFP	64 BGA	64 LQFP	48 QFN	32 QFN	引脚名	默认功能	ALT0	ALT1	ALT2	ALT3	ALT4	ALT5	ALT6	ALT7
K11	51	G8	33	25	18	PTA19	XTAL0	XTAL0	PTA19		UART1_TX	TPM_CLKIN1		LPTMP0_ALT1	
J11	52	F8	34	26	19	PTA20	RESET_b		PTA20						RESET_b
G11	53	F7	35	27	20	PTB0/LLWU_P5	ADC0_SE8/ TSI0_CH0	ADC0_SE8/ TSI0_CH0	PTB0/LLWU_P5	I2C0_SCL	TPM1_CH0				
G10	54	F6	36	28	21	PTB1	ADC0_SE9/ TSI0_CH6	ADC0_SE9/ TSI0_CH6	PTB1	I2C0_SDA	TPM1_CH1				
G9	55	E7	37	29	—	PTB2	ADC0_SE12/ TSI0_CH7	ADC0_SE12/ TSI0_CH7	PTB2	I2C0_SCL	TPM2_CH0				
G8	56	E8	38	30	—	PTB3	ADC0_SE13/ TSI0_CH8	ADC0_SE13/ TSI0_CH8	PTB3	I2C0_SDA	TPM2_CH1				
E11	57	—	—	—	—	PTB7	DISABLED		PTB7						
D11	58	—	—	—	—	PTB8	DISABLED		PTB8	SPI1_PCS0	EXTRG_IN				
E10	59	—	—	—	—	PTB9	DISABLED		PTB9	SPI1_SCK					
D10	60	—	—	—	—	PTB10	DISABLED		PTB10	SPI1_PCS0					
C10	61	—	—	—	—	PTB11	DISABLED		PTB11	SPI1_SCK					
B10	62	E6	39	31	—	PTB16	TSI0_CH9	TSI0_CH9	PTB16	SPI1_MOSI	UART0_RX	TPM_CLKIN0	SPI1_MISO		
E9	63	D7	40	32	—	PTB17	TSI0_CH10	TSI0_CH10	PTB17	SPI1_MISO	UART0_TX	TPM_CLKIN1	SPI1_MOSI		
D9	64	D6	41	—	—	PTB18	TSI0_CH11	TSI0_CH11	PTB18		TPM2_CH0	I2S0_TX_BCLK			
C9	65	C7	42	—	—	PTB19	TSI0_CH12	TSI0_CH12	PTB19		TPM2_CH1	I2S0_TX_FS			
F10	66	—	—	—	—	PTB20	DISABLED		PTB20					CMP0_OUT	
F9	67	—	—	—	—	PTB21	DISABLED		PTB21						
F8	68	—	—	—	—	PTB22	DISABLED		PTB22						
E8	69	—	—	—	—	PTB23	DISABLED		PTB23						
B9	70	D8	43	33	—	PTC0	ADC0_SE14/ TSI0_CH13	ADC0_SE14/ TSI0_CH13	PTC0		EXTRG_IN	audioUSB_ SOF_OUT	CMP0_OUT	I2S0_TXD0	
D8	71	C6	44	34	22	PTC1/LLWU_P6/ RTC_CLKIN	ADC0_SE15/ TSI0_CH14	ADC0_SE15/ TSI0_CH14	PTC1/LLWU_P6/ RTC_CLKIN	I2C1_SCL		TPM0_CH0		I2S0_TXD0	
C8	72	87	45	35	23	PTC2	ADC0_SE11/ TSI0_CH15	ADC0_SE11/ TSI0_CH15	PTC2	I2C1_SDA		TPM0_CH1		I2S0_TX_FS	
B8	73	C8	46	36	24	PTC3/LLWU_P7	DISABLED		PTC3/LLWU_P7		UART1_RX	TPM0_CH2	CLKOUT	I2S0_TX_CLK	
F7	74	E3	47	—	—	VSS	VSS	VSS							

续表

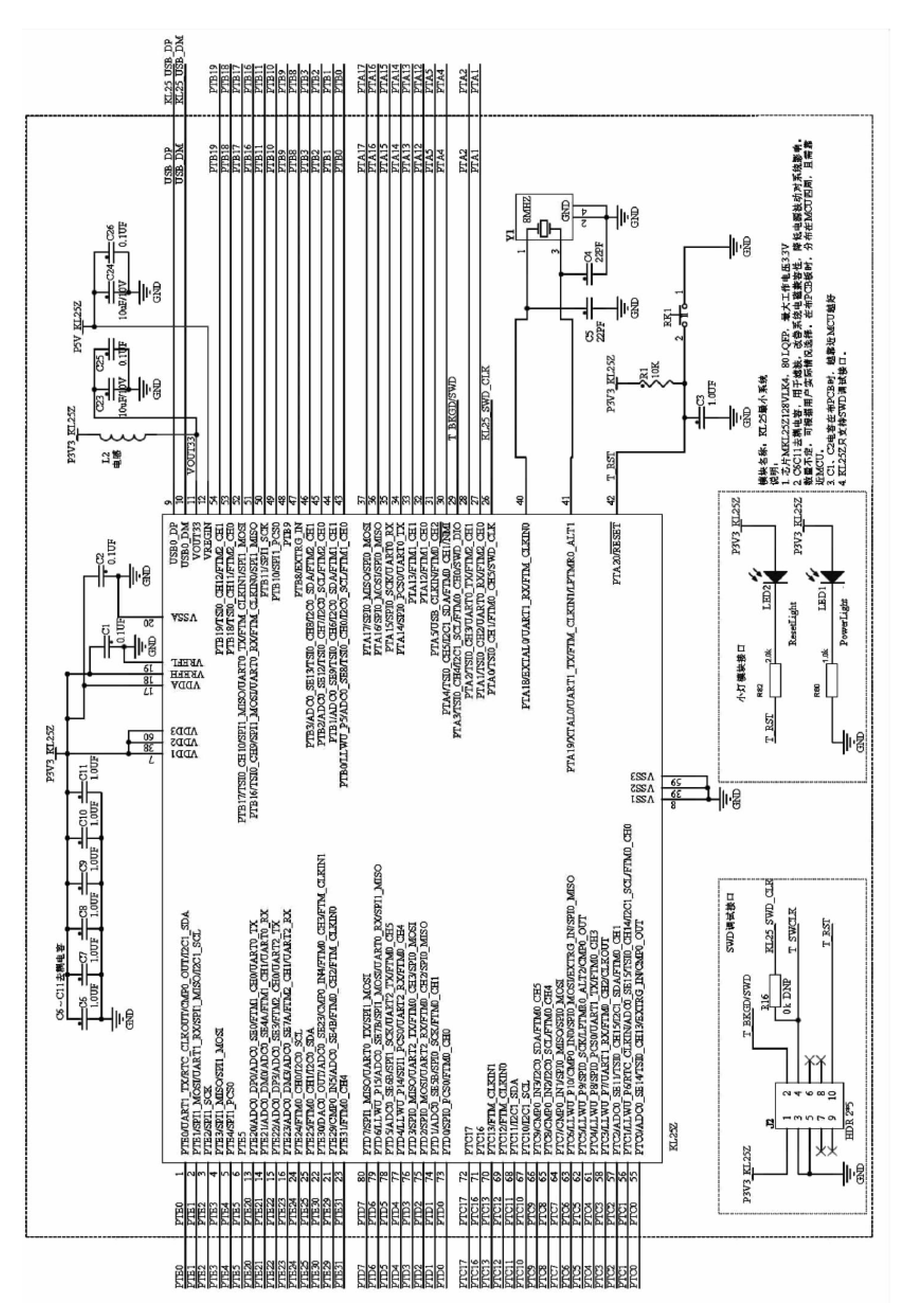
121 BGA	100 LQFP	64 BGA	64 LQFP	48 QFN	32 QFN	引脚名	默认功能	ALT0	ALT1	ALT2	ALT3	ALT4	ALT5	ALT6	ALT7
E7	75	E4	48	—	—	VDD	VDD	VDD							
A10	—	—	—	—	—	PTC20	DISABLED		PTC20						
A9	—	—	—	—	—	PTC21	DISABLED		PTC21						
B11	—	—	—	—	—	PTC22	DISABLED		PTC22						
C11	—	—	—	—	—	PTC23	DISABLED		PTC23						
A8	76	88	49	37	25	PTC4/LLWU_P8	DISABLED		PTC4/LLWU_P8	SPI0_PCS0	UART1_TX	TPM0_CH3	I2S0_MCLK		
D7	77	A8	50	38	26	PTC5/LLWU_P9	DISABLED		PTC5/LLWU_P9	SPI0_SCK	LPTMR0_ALT2	I2S0_RXD0		CMP0_OUT	
C7	78	A7	51	39	27	PTC6/LLWU_P10	CMP0_IN0	CMP0_IN0	PTC6/LLWU_P10	SPI0_MOSI	EXTRG_IN	I2S0_RX_BCLK	SPI0_MISO	I2S0_MCLK	
B7	79	B6	52	40	28	PTC7	CMP0_IN1	CMP0_IN1	PTC7	SPI0_MISO	audioUSB_ SOF_OUT	I2S0_RX_FS	SPI0_MOSI		
A7	80	A6	53	—	—	PTC8	CMP0_IN2	CMP0_IN2	PTC8	I2C0_SCL	TPM0_CH4	I2S0_MCLK			
D6	81	85	54	—	—	PTC9	CMP0_IN3	CMP0_IN3	PTC9	I2C0_SDA	TPM0_CH5	I2S0_RX_BCLK			
C6	82	84	55	—	—	PTC10	DISABLED		PTC10	I2C1_SCL		I2S0_RX_FS			
C5	83	A5	56	—	—	PTC11	DISABLED		PTC11	I2C1_SDA		I2S0_RXD0			
B6	84	—	—	—	—	PTC12	DISABLED		PTC12			TPM_CLKIN0			
A6	85	—	—	—	—	PTC13	DISABLED		PTC13			TPM_CLKIN1			
D5	90	—	—	—	—	PTC16	DISABLED		PTC16						
C4	91	—	—	—	—	PTC17	DISABLED		PTC17						
B4	92	—	—	—	—	PTC18	DISABLED		PTC18						
D4	93	C3	57	41	—	PTD0	DISABLED		PTD0	SPI0_PCS0		TPM0_CH0			
D3	94	A4	58	42	—	PTD1	ADC0_SE5b	ADC0_SE5b	PTD1	SPI0_SCK		TPM0_CH1			
C3	95	C2	59	43	—	PTD2	DISABLED		PTD2	SPI0_MOSI	UART2_RX	TPM0_CH2	SPI0_MISO		
B3	96	83	60	44	—	PTD3	DISABLED		PTD3	SPI0_MISO	UART2_TX	TPM0_CH3	SPI0_MOSI		
A3	97	A3	61	45	28	PTD4/LLWU_P14	DISABLED		PTD4/LLWU_P14	SPI1_PCS0	UART2_RX	TPM0_CH4			
A2	99	C1	62	46	30	PTD5	ADC0_SE8b	ADC0_SE8b	PTD5	SPI1_SCK	UART2_TX	TPM0_CH5			
B2	99	B2	63	47	31	PTD6/LLWU_P15	ADC0_SE7b	ADC0_SE7b	PTD6/LLWU_P15	SPI1_MOSI	UART0_RX		SPI1_MISO		
A1	100	A2	64	48	32	PTD7	DISABLED		PTD7	SPI1_MISO	UART0_TX		SPI1_MOSI		
A11	55	C5	—	—	—	NC	NC	NC							
—	87	—	—	—	—	NC	NC	NC							
—	88	—	—	—	—	NC	NC	NC							
—	89	—	—	—	—	NC	NC	NC							

备注：BGA 封装还有 J3、H3、K4、L7、J9、J4、H11、F11、A5、B5、A4、B1、C2、C1、D2、D1、E1 等引脚，全为空(NC)。

附录 B　KL25/26 硬件最小系统原理图

B.1　KL25 硬件最小系统原理图

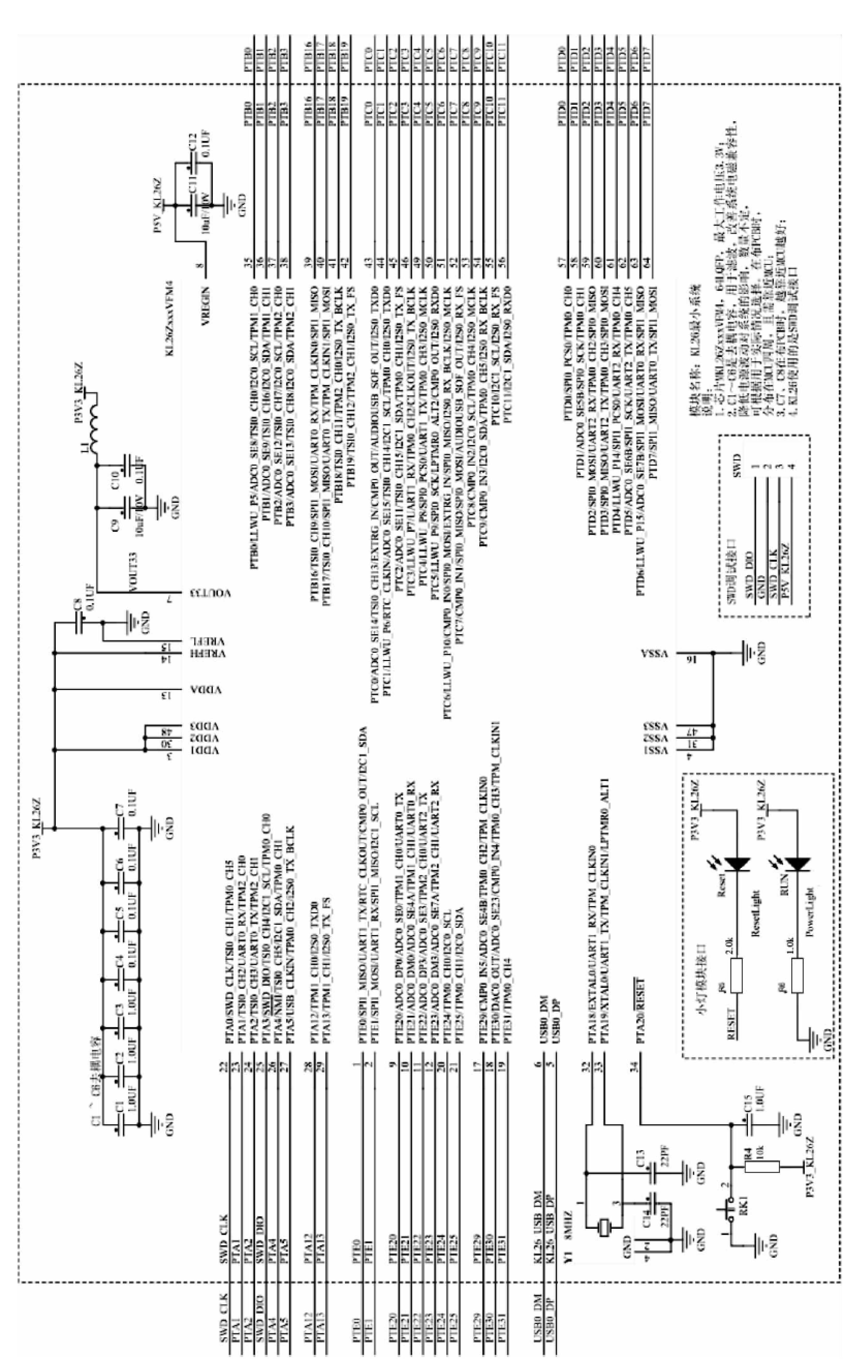
KL25 硬件最小系统原理图如附图 B-1 所示。



附图 B-1　KL25 硬件最小系统原理图

B.2　KL26 硬件最小系统原理图

KL26 硬件最小系统原理图如附图 B-2 所示。



附图 B-2　KL26 硬件最小系统原理图

附录 C　printf 的常用格式

C.1　printf 调用的一般格式

printf 函数是一个标准库函数，它的函数原型在头文件“stdio. h”中。但作为一个特例，不要求在使用 printf 函数之前必须包含 stdio. h 文件。

printf 函数调用的一般形式为：

```
printf("格式控制字符串",输出表列)
```

其中，格式控制字符串用于指定输出格式。格式控制串可由格式字符串和非格式字符串两种组成。格式字符串是以％开头的字符串，在％后面跟有各种格式字符，以说明输出数据的类型、形式、长度、小数位数等。例如：

“％d”表示按十进制整型输出；

“％ld”表示按十进制长整型输出；

“％c”表示按字符型输出等。

非格式字符串原样输出，在显示中起提示作用。输出表列中给出了各个输出项，要求格式字符串和各输出项在数量和类型上应该一一对应。

C.2　格式字符串

格式字符串的一般形式为：

[标志][输出最小宽度][精度][长度]类型

其中，方括号[]中的项为可选项。以下说明各项的意义。

1. 类型

类型字符用以表示输出数据的类型，其格式符和意义如附表 C-1 所示。

附表 C-1　类型字符

格式字符	意　　义
d	以十进制形式输出带符号整数(正数不输出符号)
o	以八进制形式输出无符号整数(不输出前缀 0)
x,X	以十六进制形式输出无符号整数(不输出前缀 0x)
u	以十进制形式输出无符号整数
f	以小数形式输出单、双精度实数

续表

格式字符	意　　义
e,E	以指数形式输出单、双精度实数
g,G	以%f或%e中较短的输出宽度输出单、双精度实数
c	输出单个字符
s	输出字符串

2. 标志

标志字符为一、十、# 和空格4种,其意义如附表C-2所示。

附表 C-2　标志字符

标　　志	意　　义
−	结果左对齐,右边填空格
+	输出符号(正号或负号)

3. 输出最小宽度

用十进制整数来表示输出的最少位数。若实际位数多于定义的宽度,则按实际位数输出,若实际位数少于定义的宽度则补以空格或0。

4. 精度

精度格式符以“.”开头,后跟十进制整数。本项的意义是:如果输出数字,则表示小数的位数;如果输出字符,则表示输出字符的个数;若实际位数大于所定义的精度数,则截去超过的部分。

5. 长度

长度格式符为h、l两种,h表示按短整型量输出,l表示按长整型量输出。

C.3　输出格式举例

```
char c,s[20];
int a;
float f;
double x;
a=1234;
f=3.14159322;
x=0.123456789123456789;
c='A';
strcpy(s,"Hello,World");
uart_init(UART_Debug,9600);              //串口2使用总线时钟20MHz
printf("苏州大学嵌入式实验室 printf 构件测试用例!\n");
//整数类型数据输出测试
printf("整型数据输出测试:\n");
printf("整数 a=%d\n", a);                //按照十进制整数格式输出,显示 a=1234
printf("整数 a=%d%%\n", a);              //输出%号结果 a=1234%
```

```
printf("整数 a=%6d\n", a);                    //输出 6 位十进制整数左边补空格,显示 a= 1234
printf("整数 a=%06d\n", a);                   //输出 6 位十进制整数左边补 0,显示 a=001234
printf("整数 a=%2d\n", a);                    //a 超过两位,按实际输出 a=1234
printf("整数 a=%-6d\n", a);                   //输出 6 位十进制整数右边补空格,显示 a=1234
printf("\n");
//浮点数类型数据输出测试
printf("浮点型数据输出测试:\n");
printf("浮点数 f=%f\n", f);                   //浮点数有效数字是 7 位,结果 f=3.14159297
printf("浮点数 fhavassda = %6.4f\n", f);      //输出 6 列,小数点后 4 位,结果 f=3.1416
printf("double 型数 x=%lf\n", x);             //输出长浮点数 x=0.12345678912345678
printf("double 型数 x=%18.15lf\n", x);        //输出 18 列,小数点后 15 位,x=0.123456789123456
printf("\n");
//字符类型数据输出测试
printf("字符类型数据输出测试:\n");
printf("字符型 c=%c\n", c);                   //输出字符 c=A
printf("ASCII 码 c=%x\n", c);                 //以十六进制输出字符的 ASCII 码 c=41
printf("字符串 s[]=%s\n", s);                 //输出数组字符串 s[]=Hello,World
printf("字符串 s[]=%6.9s\n", s);              //输出最多 9 个字符的字符串 s[]=Hello,World
```

参考文献

［1］ NXP. KL25 Sub-Family Reference Manual Rev. 3，2012.（简称 KL25 参考手册）
［2］ NXP. KL25 Sub-Family Data Sheet Rev. 3，2012.（简称 KL25 数据手册）
［3］ NXP. Kinetis L Peripheral Module Rev. 0，2012.（简称 KL25 外设快速参考手册）
［4］ NXP. Kinetis Design Studio V3.0.0-User's Guide Rev 3.0，2015（简称 KDS 用户指南）
［5］ NXP. KL26 Sub-Family Reference Manual，2015.（简称 KL26 参考手册）
［6］ NXP. KL26 Sub-Family Data Sheet Rev. 3，2014.（简称 KL26 数据手册）
［7］ NXP. Kinetis SDK V1.1.0 For KL26，2015.（简称 KL26 软件开发工具包）
［8］ NXP. FRDMKL26ZUser Guide，2013.（简称 KL26 用户指南）
［9］ ARM. Cortex-M0＋ Technical Reference Manual Rev. r0p1，2012.（简称 M0＋参考手册）
［10］ ARM. Cortex-M0＋ Devices Generic User Guide，2012.（简称 M0＋用户指南）
［11］ ARM. ARMv6-M Architecture Reference Manual，2010.
［12］ ARM. ARMv7-M Architecture Reference Manual，2010.
［13］ Joseph Yiu. The Definitive Guide to the ARM Cortex-M0. Elsevier Inc，2011.（简称 CM0 权威指南）
［14］ Joseph Yiu. Cortex-M3 权威指南（中文版）. 宋岩，译. 北京：北京航空航天大学出版社，2011.（简称 Cortex-M3 权威指南）
［15］ Free Software Foundation Inc. Using as The gnu Assembler Version 2.11.90，2012.（简称 GNU 汇编语法）
［16］ NATO Communications and Information Systems Agency. NATO Standard for Development of Reusable Software Components，1991.（简称 NATO）
［17］ 边海龙，贾少华. USB 2.0 设备的设计与开发. 北京：人民邮电出版社，2004.
［18］ ［美］Jack Ganssle，Michael Barr. 英汉双解嵌入式系统词典. 马广云，等译. 北京：北京航空航天大学出版社，2006.
［19］ ［美］Colin Walls. 嵌入式软件概论. 沈建华，译. 北京：北京航空航天大学出版社，2007.
［20］ ［美］Jack Ganssle. 嵌入式系统设计的艺术（英文版·第 2 版）. 北京：人民邮电出版社，2009.

图 书 资 源 支 持

感谢您一直以来对清华版图书的支持和爱护。为了配合本书的使用，本书提供配套的素材，有需求的用户请到清华大学出版社主页(http://www.tup.com.cn)上查询和下载，也可以拨打电话或发送电子邮件咨询。

如果您在使用本书的过程中遇到了什么问题，或者有相关图书出版计划，也请您发邮件告诉我们，以便我们更好地为您服务。

我们的联系方式：

地　　址：北京海淀区双清路学研大厦 A 座 707

邮　　编：100084

电　　话：010－62770175－4604

资源下载：http://www.tup.com.cn

电子邮件：weijj@tup.tsinghua.edu.cn

QQ：883604(请写明您的单位和姓名)

扫一扫

资源下载、样书申请

新书推荐、技术交流

用微信扫一扫右边的二维码，即可关注清华大学出版社公众号“书圈”。